Final Practice Exam Solution

Created by Ajay Gandecha and Jade Keegan for
COMP 301: Foundations of Programming
Spring 2023, Section 001

April 25th, 2023

Name: _____

This practice exam consists of 61 points, 7 questions, and 12 pages.
Note that this practice exam is shorter than the final exam.

BEWARE:
This is practice exam is not comprehensive.
The structure, content, and type of questions
found on the actual exam may differ.
This practice exam was written *before* the actual exam was written.
This practice exam is meant to be a good study tool but not your
primary method of studying. Please make sure to study just as
you would if this practice exam did not exist.
Good luck!

Question 1: True / False (18 points) Completely fill in the bubble next to your answer.

1.1. (1 point) A mutable object will never have getter methods defined.
○ True ● False

1.2. (1 point) A subinterface may extend multiple parent interfaces.
● True ○ False

1.3. (1 point) Fields declared in an interface may be public, protected, or private.
○ True ● False

1.4. (1 point) Abstract classes, like interfaces, must not contain a constructor.
○ True ● False

1.5. (1 point) Primitive types may be used when creating an object that expects a generic type.
○ True ● False

1.6. (1 point) One subclass of a parent class has an is-a relationship with another class that extends the parent class.
○ True ● False

1.7. (1 point) If a method has a `return` in the `try`/`catch` block, the `finally` block will not necessarily run.
○ True ● False

1.8. (1 point) Different versions of an overloaded method must be distinguished by having different return types.
○ True ● False

1.9. (1 point) A class that employs the Factory design pattern will generally have public constructors.
○ True ● False

1.10. (1 point) Multiple iterators may be traversing a particular iterable collection at a time.
● True ○ False

1.11. (1 point) A lambda expression is a more concise syntax for implementing the strategy design pattern.
● True ○ False

1.12. (1 point) In the observer design pattern, all classes that act as observers of a particular subject class must implement the same interface.
● True ○ False

1.13. (1 point) The multiton design pattern can be useful for dynamically determining which of several different subclasses to use at runtime.
○ True ● False

1.14. (1 point) An object that supports dependency injection will typically have a constructor with parameters for specifying the object's component parts.
● True ○ False

1.15. (1 point) Multiple threads may not simultaneously execute a synchronized method of two different objects of the same type.
○ True ● False

1.16. (1 point) The join() method of a Thread object can be used to temporarily pause the execution of that thread.
● True ○ False

1.17. (1 point) A program that has been parallelized to use 20 threads can be expected to run about twenty times as fast as a program parallelized to use 1 thread when running on a machine that has 4 cores.
○ True ● False

1.18. (1 point) In JavaFX, a scene object represents the display window, while a stage object is a container for the component tree that will be displayed in the window.
○ True ● False

Question 2: Elements of a Java Class (4 points) Consider the following code.

```
1  public class Lynel {
2
3    private static final int POWER = 50;
4
5    private int xPosition;
6    private int yPosition;
7
8    public Lynel(int x, int y) {
9      xPosition = x;
10     yPosition = y;
11   }
12
13   public int getX() {
14     return this.xPosition;
15   }
16
17   public int getY() {
18     return this.yPosition;
19   }
20
21   public int getXOffset(int o) {
22     return this.getX() + o;
23   }
24
25   public int getYOffset(int o) {
26     return this.getY() + o;
27   }
28
29   public static int getPower() {
30     return POWER;
31   }
32 }
```

2.1. (1 point) List the names of all class members defined by `Foo`.

Solution: `POWER`, `getPower()`

2.2. (1 point) List the names of all instance members defined by `Foo`.

Solution: `xPosition`, `yPosition`, `getX()`, `getY()`, `getXOffset()`, `getYOffset()`

2.3. (1 point) List the names of all fields defined by `Foo`.

Solution: `POWER`, `xPosition`, `yPosition`

2.4. (1 point) List the names of all methods defined by `Foo`.

Solution: `getX()`, `getY()`, `getXOffset()`, `getYOffset()`, `getPower()`

Question 3: Virtual Method Execution (5 points) Consider the following Java class definitions.

```
1  class A {
2    public int calculate(int x, int y) {
3      return x + 4 * y + 10;
4    }
5  }
6
7  class B extends A {
8    @Override
9    public int calculate(int x, int y) {
10     return x * y + 2;
11   }
12 }
```

3.1. (1 point) What is the output of the following code snippet if `calculate()` is virtual?

```
1 B b = new B();
2 System.out.print(b.calculate(0, 1));
```

Solution: `2`

3.2. (1 point) What is the output of the following code snippet if `calculate()` is **not** virtual?

```
1 B b = new B();
2 System.out.print(b.calculate(2, 3));
```

Solution: `8`

3.3. (1 point) What is the output of the following code snippet if `calculate()` is virtual?

```
1 A a = new B();
2 System.out.print(a.calculate(7, 9));
```

Solution: `65`

3.4. (1 point) What is the output of the following code snippet if `calculate()` is **not** virtual?

```
1 A a = new B();
2 System.out.print(a.calculate(10, 2));
```

Solution: `53`

3.5. (1 point) In Java, is `calculate()` virtual?

- ● Yes, virtual
- ○ No, not virtual

Question 4: Exception Code Tracing (4 points) Use the following `Exception` class definitions below to solve the following problems.

```
1 class ExA extends Exception {}
2 class ExB extends RuntimeException {}
3 class ExC extends IOException {}
4 class ExD extends ExA {}
5 class ExE extends ExC {}
6 class ExF extends ExB {}
7 class ExG extends ExE {}
8 class ExH extends ExG {}
```

4.1. (2 points) Write all of the subclasses are subject to the "catch or specify" policy.

Solution:
`Exception`, `ExA`, `IOException`, `ExC`, `ExD`, `ExE`, `ExG`, `ExH`

4.2. (2 points) Given `ExH`, write down all of the possible `catch` statements that could catch the exception if thrown within a `try` block.

Solution:

```
catch(ExH e)
catch(ExG e)
catch(ExE e)
catch(ExC e)
catch(IOException e)
catch(Exception e)
```

Question 5: Writing Code To specification (10 points) Write a simple `GameBoy` class that models a Nintendo GameBoy console with the following features:

- A `GameBoy` object should have the following immutable read-only properties:
  - The color of the GameBoy (as a string).
  - The version of the GameBoy (as a string).
- A `GameBoy` object should have the following mutable (i.e., changeable) properties:
  - The current battery percentage of the GameBoy (0.0-1.0).
  - The current game inserted into the GameBoy (as a string).
- You should provide three different forms of constructor:
  - One for which all values are specified as parameters.
  - One which assumes the GameBoy is new (assume new consoles have a full battery, and that the GameBoy is a blue GameBoy Advance. Let the game be specified as a parameter). You MUST use constructor chaining to implement this method.
  - One which assumes that your older sibling just gave you their old green GameBoy Color as a hand-me-down, with a battery level at `0.301` and the game inserted as "The Legend of Zelda: Link's Awakening". You MUST use constructor chaining to implement this method.

Please use the common Java conventions for getters and setters.

Your `GameBoy` class:

Solution:

```
public interface GameBoy {

    // Immutable Fields
    private final String color;
    private final String version;

    // Mutable Fields
    private double battery;
    private String game;

    // Default Constructor
    // NOTE: Use this for constructor chaining
    public GameBoy(String color, String version, double battery, String
        game) {
        this.color = color;
        this.version = version;
        this.battery = battery;
        this.game = game;
    }

    // Hand-me-down Constructor
    public GameBoy() {
        this("green", "GameBoy Color", 0.301, "The Legend of Zelda: Link's
            Awakening");
    }

    // New Console Constructor
    public GameBoy(String game) {
        this("blue", "GameBoy Advance", 1.0, game);
    }

    // Getters
    public String getColor() { return this.color; }
    public String getVersion() { return this.version; }
    public double getBattery() { return this.battery; }
    public String getGame() { return this.game; }

    // Setters
    // NOTE: Must only exist for mutable fields.
    public void setBatteryLevel(double new) {
        if(new < 0.0 || new > 1.0) {
            throw new IllegalArgumentException();
        }
        this.battery = new;
    }

    public void setGame(String new) {
        this.game = new;
    }
}
```

Question 6: Inheritance (20 points) Refactor the code given below for the classes `Lizalfos`, `Stal` and `Molduga` using inheritance as appropriate such that:

- `Lizalfos` and Stal are subclasses of a common parent class called `Mob`
- `Mob` is a subclass of a parent class called `Enemy`.
- All `Enemy` objects encapsulate hit points and power. They also include corresponding getters and other methods to calculate damage received/dealt.
- The enumeration `AttackType` is defined as follows in the (not provided) class `Player`: public enum AttackType REGULAR, FIRE, ICE, ELECTRIC

*Note that the implementation of these classes are significantly simplified for the sake of brevity and are not accurate to Breath of the Wild. :')

```
1  public class Lizalfos {
2      private int hit_points, power;
3      private String weapon;
4      private Variation variation;
5
6      public enum Variation { REGULAR, BLUE, BLACK, SILVER, GOLDEN, FIREBREATH,
           ICEBREATH, ELECTRIC }
7
8      public Lizalfos(int hit_points, int power, Variation variation) {
9          this.hit_points = hit_points;
10         this.power = power;
11         this.weapon = weapon;
12         this.variation = variation;
13     }
14
15     public int getHitPoints() { return hit_points; }
16
17     public int getPower() { return power; }
18
19     public int getVariation() { return var; }
20
21     public String getWeapon() { return weapon; }
22
23     public void setWeapon(String weapon) { this.weapon = weapon; }
24
25     public int calculateDamageReceived(AttackType attack) {
26         if (variation == Variation.FIREBREATH) {
27             if (attack == AttackType.ICE) { hit_points -= hit_points/8; }
28             else if (attack == AttackType.FIRE) { hit_points -= hit_points
                   /16; }
29             else { hit_points -= hit_points/12; }
30         } else if (variation == Variation.ICBREATH) { // Implementation }
31         // ... Further Implementation Based on Each Lizalfos Variation
32     }
33
34     public int getDamageDealt() {
35         if (weapon != null) {
36             if (variation != Variation.REGULAR) { return power + 15; }
37             else { return power + 5; }
38         } else { return power; }
39         }
40     }
41 } // End of Lizalfos
```

```java
public class Stal {
    private int hit_points, power;
    private String weapon;

    public Stal(int hit_points, int power, Variation variation, String weapon
        ) {
        this.hit_points = hit_points;
        this.power = power;
        this.weapon = weapon;
    }

    public int getHitPoints() { return hit_points; }

    public int getPower() { return power; }

    public String getWeapon() { return weapon; }

    public void setWeapon(String weapon) { this.weapon = weapon; }

    public int calculateDamageReceived(AttackType attack) {
        if (attack != AttackType.REGULAR) { hit_points -= hit_points/8; }
        else { hit_points -= hit_points/12; }
    }

    public int getDamageDealt() {
        if (weapon != null) { return power + 5; }
        else { return power; }
    }
} // End of Stal
```

Fill in the restructured code in the blanks on the following pages. Make sure your newly written code has all the same functionality as the original code!

Your code for `Enemy`:

Solution:

```
abstract public class Enemy {
    private int hit_points;
    private int power;

    public Enemy(int hit_points, int power) {
        this.hit_points;
        this.power;
    }

    public int getHitPoints() { return hit_points; }

    public int getPower() { return power; }

    abstract public int calculateDamageReceived(AttackType attack);

    abstract public int getDamageDealt();

} // End of Enemy
```

Your code for `Mob`:

Solution:

```
abstract public class Mob extends Enemy {
    private String weapon;

    public Mob(int hit_points, int power, String weapon) {
        super(hit_points, power);
        this.weapon = weapon;
    }

    abstract public int calculateDamageReceived(AttackType attack);

    abstract public int getDamageDealt();

} // End of Mob
```

Your code for `Lizalfos`:

Solution:

```
public class Lizalfos extends Mob {
    private Variation variation;

    public enum Variation { REGULAR, BLUE, BLACK, SILVER, GOLDEN,
       FIREBREATH, ICEBREATH, ELECTRIC }

    public Lizalfos(int hit_points, int power, String weapon, Variation
       variation) {
        super(hit_points, power, weapon);
        this.variation = variation;
    }

    @Override
    public int calculateDamageReceived(AttackType attack) {
    if (variation == Variation.FIREBREATH) {
        if (attack == AttackType.ICE) { hit_points -= hit_points/8; }
        else if (attack == AttackType.FIRE) { hit_points -= hit_points/16;
            }
        else { hit_points -= hit_points/12; }
    } else if (variation == Variation.ICBREATH) {
        // ... Further Implementation Based on Each Lizalfos Variation
    }

    @Override
    public int getDamageDealt() {
        if (weapon != null) {
            if (variation != Variation.REGULAR) { return power + 15; }
            else { return power + 5; }
        } else { return power; }
        }
    }

} // End of Lizalfos
```

Your code for `Stal`:

Solution:

```
public class Stal extends Mob {
    public Stal(int hit_points, int power, String weapon) {
        this.hit_points = hit_points;
        this.power = power;
        this.weapon = weapon;
    }

    @Override
    public int calculateDamageReceived(AttackType attack) {
        if (attack != AttackType.REGULAR) { hit_points -= hit_points/8; }
        else { hit_points -= hit_points/12; }
    }

    @Override
    public int getDamageDealt() {
        if (weapon != null) { return power + 5; }
        else { return power; }
    }
} // End of Stal
```

Question 7: Further Practice (0 points) These are more code writing questions you can try to construct yourself and practice with! We will *not* be providing the code. These are just ideas to help you self-study. Good luck!

Determine which Design Pattern to use and implement the following (including relevant methods):

- A `DivineBeastScourge` object should have the following properties:
  - Exactly four unique instances of the Divine Beast Scourge boss monsters, including `WaterblightGanon`, `FireblightGanon`, `WindblightGanon`, and `ThunderblightGanon`.
  - The power level (base damage dealt) of the boss.
  - The hit points of the boss.
  - The location of the boss.
  - *Hint:* You should not make subclasses for this!
- A `MasterSword` object should have the following properties:
  - There should only be one instance of the `MasterSword` object.
  - The durability of the weapon.
  - The power level (base damage dealt) of the weapon.
- A `Costumizer` object should have the following properties:
  - Implements `Character`, which represents Link (the in-game character) and has a method that prints out the character's information.
  - Used to add a costume to Link (the `Character`).
  - Contains a method that prints out text with the character's name (Link) and describes the costume. (Ex: "Link wearing the Desert Voe Set")
- A `VillagerIterator` object should have the following properties:
  - Iterates through the list of `Villager` objects in a `Village`.