Final Practice Exam

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.

```java
public class Lynel {

  private static final int POWER = 50;

  private int xPosition;
  private int yPosition;

  public Lynel(int x, int y) {
    xPosition = x;
    yPosition = y;
  }

  public int getX() {
    return this.xPosition;
  }

  public int getY() {
    return this.yPosition;
  }

  public int getXOffset(int o) {
    return this.getX() + o;
  }

  public int getYOffset(int o) {
    return this.getY() + o;
  }

  public static int getPower() {
    return POWER;
  }
}
```

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

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

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

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

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

```java
class A {
  public int calculate(int x, int y) {
    return x + 4 * y + 10;
  }
}

class B extends A {
  @Override
  public int calculate(int x, int y) {
    return x * y + 2;
  }
}
```

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

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

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

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

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

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

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

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

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.

```
class ExA extends Exception {}
class ExB extends RuntimeException {}
class ExC extends IOException {}
class ExD extends ExA {}
class ExE extends ExC {}
class ExF extends ExB {}
class ExG extends ExE {}
class ExH extends ExG {}
```

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

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.

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:

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. :')

```java
public class Lizalfos {
    private int hit_points, power;
    private String weapon;
    private Variation variation;

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

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

    public int getHitPoints() { return hit_points; }

    public int getPower() { return power; }

    public int getVariation() { return var; }

    public String getWeapon() { return weapon; }

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

    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) { // Implementation }
        // ... Further Implementation Based on Each Lizalfos Variation
    }

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

Your code for `Mob`:

Your code for `Lizalfos`:

Your code for `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`.