Midterm m02 Practice Exam Solution

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

March 23, 2023

Name: ______________________________

We recommend that you set a timer for 75 minutes to complete this practice exam,
which consists of 65 points, 6 questions, and 13 pages.
Feel free to either complete this practice exam with or without the timer.
Note that on exam day, you will have 75 minutes.

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 (20 points) Completely fill in the bubble next to your answer.

1.1. (1 point) A subclass can implement multiple interfaces, while a subclass can only extend a single abstract class.
[✓ True] False

1.2. (1 point) Abstract classes can be instantiated directly, while interface cannot.
True [✓ False]

1.3. (1 point) Abstract classes allow for the use of access modifiers (`public`, `private`, etc.), while interfaces do not.
[✓ True] False

1.4. (1 point) Abstract classes can include fields and constructors, while interfaces cannot.
[✓ True] False

1.5. (1 point) An error can either be a compile time error or a runtime error, but not both.
[✓ True] False

1.6. (1 point) A program might terminate due to an uncaught checked exception.
True [✓ False]

1.7. (1 point) JUnit is a library to help write integration tests.
True [✓ False]

1.8. (1 point) The expression `assertEquals(a, b)` is always equal to `assertTrue(a == b)`.
True [✓ False]

1.9. (1 point) The `finally` block in a try-catch cannot execute unless an exception is caught.
True [✓ False]

1.10. (1 point) Only one `catch` block can be executed in a single try-catch block.
[✓ True] False

1.11. (1 point) The `for-each` loop allows us to iterate over a collection in a more concise way than a traditional `for` loop.
[✓ True] False

1.12. (1 point) The `for-each` loop provides access to the current index of an iteration.
True [✓ False]

1.13. (1 point) Classes implementing the *Singleton* design pattern should include public constructors.
True [✓ False]

1.14. (1 point) We can traverse an iterable collection numerous times with different iterators.
[✓ True] False

1.15. (1 point) In order to create anonymous classes, we must create a new Java file.
True [✓ False]

1.16. (1 point) In the *Singleton* design pattern, a instance method calls on the Singleton constructor to run.
True [✓ False]

1.17. (1 point) Calling `.hasNext()` on an `Iterator<>` object may result in a `NoSuchElementException` to be thrown.
True [✓ False]

1.18. (1 point) We can use the *Factory* design pattern to dynamically create instances of subclasses.
[✓ True] False

1.19. (1 point) In memory, the *Decorator* design pattern resembles a tree data structure.
True [✓ False]

1.20. (1 point) In the *Observer* design pattern, the subject maintains a list of observers.
[✓ True] False

Question 2: Multiple Choice (5 points) Completely fill in the bubble next to your answer using a pencil. Each question should have exactly one filled-in bubble.

2.1. (1 point) Which of the following is a *compile-time error*?

- ○ Running out of heap memory
- ● Returning a value in a `void` method
- ○ Attempting to divide a number by 0
- ○ Attempting to call a method on a field that is not initialized in the constructor

2.2. (1 point) Which of the following is NOT true about JUnit unit testing?

- ○ We can run `assert` statements inside of `try/catch` blocks.
- ○ We can use `assertArrayEquals` to easily compare two lists.
- ● We can test that a method returns a correct value, but not that a method throws an error.
- ○ We should write small, specific tests that test specific functionality for a given class.

2.3. (1 point) Which of the following are true about dependency injection?

- ○ It usually less code to write a structure that supports dependency injection compared to one that does not.
- ○ Dependency injection supports tight coupling.
- ● Dependency injection allows dependencies to be changed at runtime.
- ○ None of the above

2.4. (1 point) Which of the following structures allow us to concisely define an action without creating an explicitly-named method?

- ● Lamda expressions
- ○ Factory design patterns
- ○ Observer design patterns
- ○ Abstract classes

2.5. (1 point) Which of the following would be a proper registration method for a class that is the subject of an observer that implements the interface `EventHandler<CreeperExplosion>`?

- ○ `void register(CreeperExplosion ce)`
- ● `void register(EventHandler<CreeperExplosion> eh)`
- ○ `void register(CreeperExplosion<EventHandler> ce)`
- ○ `void EventHandler<CreeperExplosion> register()`
- ○ `void CreeperExplosion register(EventHandler<CreeperExplosion> eh)`

Question 3: Fill-In-The-Blank and Long Answer (14 points) Complete the table and questions.

3.1. (6 points) Complete the following table by organizing the seven design patterns into the categories of design patterns:
Bank: *Multiton*, *Observer*, *Decorator*, *Singleton*, *Iterator*, *Factory*, *Strategy*
NOTE: Not all cells will be filled.

Creational Patterns	**Structural Patterns**	**Behavioral Patterns**
Factory	Decorator	Iterator
Singleton		Observer
Multiton		Strategy

3.2. (4 points) List and briefly explain the four stages of software testing.

Solution:
1. Unit testing – during development, testing code in isolation
2. Integration testing – during development, testing integration with other code
3. System testing – after development, testing entire system functionality
4. Acceptance testing – during production / before release, final testing stages

3.3. (2 points) Explain the goal of lamda expressions and anonymous classes.

Solution:
Lambda expressions and anonymous classes allow for more concise code and eliminate the need to create a new .java file in order to create a new instance of an interface. Use cases include any implementations where code logic is not complex (ex: implementing the update() method on an Observer interface).

3.4. (2 points) Provide a real-world example in which the Observer/Observable design pattern could be successfully used.

Solution:
The Observer/Observable design pattern can be used for creating any sort of responsive system where events occurring in one part of the environment leads to responses in another. Common use cases can be found in the video game industry, for example, where players interact heavily with code logic through event triggers and responses.

Question 4: Exception Code Tracing (3 points) Use the following `Exception` class definitions below to solve the following problems.

```java
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. (1 point) Write all of the subclasses are NOT subject to the "catch or specify" policy.

Solution:
`ExB` and `ExF`

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:

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

Question 5: Code Completion (5 points) Fill in the blanks to complete the implementation of the Singleton design pattern:

```java
public class Singleton {

  /*__BLANK A__*/ Singleton instance;

  /*__BLANK B__*/ Singleton() {
      // Implementation not shown.
  }

  /*__BLANK C__*/ Singleton create() {
    if(/*__BLANK D__*/) {
      instance = new Singleton();
    }
    return /*__BLANK E__*/
  }
}
```

Fill in the blanks below:

5.1. (1 point) Blank A: `private`

5.2. (1 point) Blank B: `private`

5.3. (1 point) Blank C: `public static`

5.4. (1 point) Blank D: `instance == null`

5.5. (1 point) Blank E: `instance`

Question 6: Free Response (18 points) *Minecraft*, coincidentally created in Java, is one of the most popular video games to date. This question involves implementing features for the Minecraft game.

In order to interact with blocks and creatures in a *Minecraft* world, players must use tools - including swords, pickaxes, axes, and more. For the purposes of this question, assume that we are focusing only on the ***sword*** tool.

Swords in Minecraft come in 6 different materials: wooden, stone, iron, gold, diamond, and netherite. Each type of sword has its own associated attack damage values and durability values. The values for each type of sword are shown below:

Sword Material	Attack Damage	Durability
Wooden	5.0	60.0
Gold	5.0	33.0
Stone	6.0	132.0
Iron	7.0	251.0
Diamond	8.0	1562.0
Netherite	9.0	2032.0

We can store the different types of swords using an **enumeration** structure with the name `ToolMaterial`, and we can represent a sword using a **class**. The implementations of this structure is shown below:

```java
/** Enum that represents the different possible material tools. */
public enum ToolMaterial {
  // TODO: Question 6.1
}

/** Class to model a Sword object. */
public class Sword implements SwordInterface {

  // Material of the sword
  private ToolMaterial toolMaterial;
  // Damage dealt to an entity on use
  private double attackDamage;
  // Number of sword swings before the tool breaks
  private double durability;

  /* Initializes ‘Sword‘ fields */
  public Sword(ToolMaterial material, double atk, double dur) {
    this.toolMaterial = material;
    this.attackDamage = atk;
    this.durability = dur;
  }

  /* GETTER Methods */
  public ToolMaterial getMaterial() { return this.toolMaterial; }

  public double getAttackDamage() { return this.attackDamage; }

  public double getDurability() { return this.durability; }
}
```

6.1. (2 points) In the box below, complete the correct implementation of the `ToolMaterial` enum given the different sword materials referenced in the instructions.

Solution:

```
/** Enum that represents the different possible material tools. */
public enum ToolMaterial {
    WOODEN,
    GOLD,
    STONE,
    IRON,
    DIAMOND,
    NETHERITE
}

/* REQUIREMENTS:
  This is more of an exercise in remembering the syntax for an ‘enum‘
  in Java. These types are incredibly useful, so it is importnant to
  remember how they work and how they are written.
*/
```

6.2. (5 points) For the next part of this question, provide the correct implementation for a `SwordFactory` class which implements the factory design pattern. This class should provide methods to create swords of the six different types referenced in the instructions with the correct attack damage and durability values. You may name these methods whatever you like as long as they conform to the standards of the factory design pattern.

Solution:

```java
/** Class that builds swords of various materials */
public class SwordFactory {

  public static Sword makeWoodenSword() {
    return new Sword(ToolMaterial.WOODEN, 5.0, 60.0);
  }

  public static Sword makeGoldSword() {
    return new Sword(ToolMaterial.GOLD, 5.0, 33.0);
  }

  public static Sword makeStoneSword() {
    return new Sword(ToolMaterial.STONE, 6.0, 132.0);
  }

  public static Sword makeIronSword() {
    return new Sword(ToolMaterial.IRON, 7.0, 251.0);
  }

  public static Sword makeDiamondSword() {
    return new Sword(ToolMaterial.DIAMOND, 8.0, 1562.0);
  }

  public static Sword makeNetheriteSword() {
    return new Sword(ToolMaterial.NETHERITE, 9.0, 2032.0);
  }
}

/* REQUIREMENTS
- Each method MUST be `static` to conform to the design pattern
- Use correct values for each sword type
- Factory methods should not take in any parameters
- Factory methods should return a `Sword`
- Correct usage of the `public` and `private` keywords
*/
```

6.3. (5 points) Now that you have completed the implementation of `SwordFactory`, as any good software engineer should, you must now write tests to ensure that your class works effectively. For the sake of your wrists, you only have to write one unit test - one to test your factory's ability to create *diamond swords*.

Ensure to test for the following things:

- Ensure object was created and returned by the method.
- Ensure that the object types are correct.
- Ensure that the object field values are correct.

In the box below, complete the correct implementation of the test using `JUnit`. Assume all relevant import statements are provided.

Solution:

```
public class SwordFactoryTest {

  @Test
  public void testMakeDiamondSword() {

    // Test to ensure object is created
    Sword myDiamondSword = SwordFactory.makeDiamondSword();
    assertNotNull(myDiamondSword);

    // Ensure type is correct
    assertEquals(myDiamondSword.getMaterial(), ToolMaterial.DIAMOND);

    // Ensure fields are correct
    assertEquals(myDiamondSword.getAttackDamage(), 8.0);
    assertEquals(myDiamondSword.getDurability(), 1562.0);

  }
}

/* REQUIREMENTS
- Must use ‘assert‘ to test different cases
- Test that the sword is not null
- Test that the material is ToolMaterial.DIAMOND
- Test that the attack damage is 8.0
- Test that the durability is 1562.0
*/
```

6.4. (6 points) Minecraft tools are great by themselves. But, they are even better with *enchantments*. Enchantments added to tools *enhance* certain characteristics about that tool. For example, when we apply the *Sharpness* enchantment to a sword, we *increase the sword's attack damage*.

There are *five different levels* of the Sharpness Enchantment: Sharpness I - V. In game, each level becomes increasingly harder to obtain, but the reward is (usually) worth it.

In Minecraft, the formula `0.5 * max(0, level-1) + 1.0` specifies the **extra damage (i.e., new damage = original damage + the result of the formula)** that the sword inflicts. Assume that `level` as specified in the formula is an integer between 1 and 5.

There is also an **Unbreaking** enchantment with three levels: 1-3, which increases the *durability* of a sword. The formula for this is slightly more complex, so let's simplify it - for every level applied, the durability doubles. So, the new durability is the formula `original * ( 2 ^ level)`.

The final task of this question is to use the *Decorator* design patterns to implement two classes - `SharpnessSword` and `UnbreakingSword` respectively - which represent a sword with the enchantment applied.

For example, given a base sword `baseSword`, you can expect the following behavior:

```java
// "Applies" the sharpness level 2 enchantment to the base sword
Sword sharpSword = new SharpnessSword(baseSword, 2);
// The following should output the adjusted attack damage
System.out.println(sharpSword.getAttackDamage());

// "Applies" the unbreaking level 3 enchantment to the base sword
Sword unbreakingSword = new UnbreakingSword(baseSword, 3);
// The following should output the adjusted durability (so, durabiliy *
   8)
System.out.println(unbreakingSword.getDurability());

// Sample code applying two enchantments at once
Sword verySharpSword = new SharpnessSword(baseSword, 5);
Sword opSword = new UnbreakingSword(verySharpSword, 3);
```

Hints:

- You can use `@Override` to override the respective getter methods.
- Assume the `Math` package is imported! Use `Math.max(a,b)` for max values and `Math.pow(base,exponent)` for exponents.
- In Minecraft, usually you cannot stack the same enchantment on an item (for example, applying two Unbreaking III enchantments on one item). Do not worry about implementing this protection.

Provide the complete implementation for the ***SharpnessSword*** *class below:*

Solution:

```
public class SharpnessSword implements SwordInterface {

  // Field to encapsulate the base sword
  private Sword baseSword;
  // Enchantment level
  private int level;

  public SharpnessSword(Sword base, int level) {
    // Parameter validation
    if(Sword == null || level < 1 || level > 5) {
      throw new IllegalArgumentException();
    }

    this.baseSword = base;
    this.level = level;
  }

    @Override
    public ToolMaterial getMaterial() {
        return baseSword.getMaterial();
    }

  @Override
  public double getAttackDamage() {
    double modifier = 0.5 * Math.max(0, this.level - 1) + 1.0;
    return baseSword.getAttackDamage() + modifier;
  }

  @Override
  public double getDurability() {
    return baseSword.getDurability();
  }
}
```

Provide the complete implementation for the ***UnbreakingSword*** *class below:*

Solution:

```java
public class UnbreakingSword implements SwordInterface {

  // Field to encapsulate the base sword
  private Sword baseSword;
  // Enchantment level
  private int level;

  public UnbreakingSword(Sword base, int level) {
    // Parameter validation
    if(Sword == null || level < 1 || level > 3) {
      throw new IllegalArgumentException();
    }

    this.baseSword = base;
    this.level = level;
  }

    @Override
    public ToolMaterial getMaterial() {
        return baseSword.getMaterial();
    }

    @Override
    public double getAttackDamage() {
        return baseSword.getAttackDamage();
    }

  @Override
  public double getDurability() {
    double modifier = Math.pow(2.0, this.level);
    return baseSword.getDurability() * modifier;
  }

}
```

This box is ungraded. Use this as scratch paper:

Solution:

```
/* REQUIREMENTS FOR THE DECORATOR QUESTIONS
- Both classes must implement ‘SwordInterface‘.
- Both classes must encapsulate the base sword
- Both classes must take in the base sword and level as a parameter
- Both classes must include parameter validation
- Both classes must override the correct methods
  - Both methods must return the correct value
  - Both methods must use the base sword’s values in calculation
  - Using the ‘@Override‘ flag is not required
  - No method overloading - must return a ‘double‘
- Use ‘private‘ and ‘public‘ keywords where appropriate
*/
```