Paragas, Eruell John S. OOP Finals Assignment # 1
BSCS-2D

1. Many beginners confuse abstraction with encapsulation, and vice versa, because both hide details in different ways. In your own words, differentiate abstraction and encapsulation in terms of the following:

- What they hide and why they hide it
- How do they support each other
- Examples where it matters

Based on my research, abstraction and encapsulation are related ideas in object-oriented programming, but they are not the same. Both of them hide details, but they hide different kinds of details and for different reasons.

For what they hide and why they hide it, abstraction hides the complicated parts of how something works so that we only see the important features. It focuses on what an object can do, not how it does it. This makes things easier to understand because we don't need to deal with unnecessary internal processes. For example, when we drive a car, we only use the steering wheel, pedals, and buttons. We don't need to know how the engine burns fuel or how the transmission shifts gears because those details are not important for driving.

Encapsulation, meanwhile, hides the data inside an object. It protects the internal variables by only allowing controlled access through methods like getters and setters. This prevents accidental changes or unauthorized access to the object's internal state. In simple words, encapsulation answers the question: "How do we protect the object's data and control who can change it?"

Even though abstraction and encapsulation hide different things, they support each other. I have learned that abstraction provides a clean and simple interface that the user interacts with, while encapsulation keeps the internal workings and data safe behind that interface. If there was no encapsulation, abstraction would be useless because people could still touch or modify the inner parts directly. And if there was no abstraction, encapsulation would be harder to use because everything would be visible and confusing. So they work hand-in-hand to make code more organized, secure, and easier to maintain.

Reference (APA 7th Edition)
Code_r. (2025, July 12). *Difference between abstraction and encapsulation in Java with examples*. GeeksforGeeks.https://www.geeksforgeeks.org/java/difference-between-abstraction-and-encapsulation-in-java-with-examples/

2. In encapsulation, the buzz words associated with it are getters and setters. While this becomes a way to hide the data of an object, the approach might induce a fake sense of security. How does this happen in encapsulation and how to effectively prevent or improve its data hiding capability?

Based on my understanding of my research, in encapsulation, the terms “getters” and “setters” are often used because they allow controlled access to an object’s private data. By making class variables private and giving public methods to read or modify them, it seems like the object’s internal data is protected. However, this can sometimes give a **false sense of security**. Even though the data is technically private, public getters and setters still let other parts of the program read or change it. If the setter doesn’t have proper validation, it’s possible for external code to assign invalid or harmful values, which bypasses the intended protection of the object’s state. So, encapsulation only truly protects data when access methods are carefully designed; otherwise, it is just a superficial layer.

To make data hiding in encapsulation more effective, I found several practices from my research. First, setters should include proper validation to ensure only valid values are assigned to private variables. For example, if a variable represents a person’s age, the setter should prevent negative numbers or unrealistically high values. Second, not all variables need both getters and setters; some can be made read-only by providing only a getter, which prevents unwanted changes. Third, access to sensitive data or methods can be limited using stricter access modifiers, like protected or package-private, to restrict usage to specific classes or subclasses. Finally, designing classes to be immutable where the internal state cannot be changed after creation can further improve security and reliability.

In conclusion, getters and setters are important tools in encapsulation, but careless use can weaken data hiding. True protection comes from thoughtful design, proper validation, and selectively exposing data so that encapsulation really provides both modularity and security.

References:

Agarwal, H. (2025, November 21). *Encapsulation in Java*.
GeeksforGeeks. https://www.geeksforgeeks.org/java/encapsulation-in-java/

3. An effective way of abstraction is the use of interfaces. As a "contract" of the class, every members of the interface (attributes and methods) must be implemented when creating a class. While it may enforce a better design, it may result to issues where some behaviors in the interface cannot be fully implemented by the class. Give one scenario where it happens and provide a workaround for this.

SCENARIO I

Suppose we have an interface Vehicle in Java:

```
public interface Vehicle {

    void startEngine();

    void fly();

}
```

Now, if we want to create a truck class that implements Vehicle:

```
public class truck implements Vehicle {

    @Override

    public void startEngine() {

        System.out.println("Car engine started.");

    }

    @Override

    public void fly() {

        // truck cannot fly, but we are forced to implement this method

        System.out.println("truck cannot fly!");

    }

}
```

PROBLEM: The truck class is forced to implement the fly() method, even though a truck cannot fly. This leads to unnecessary or misleading code, one of the drawbacks of using interfaces rigidly.

WORKAROUND

Split interfaces

Instead of having a single interface with unrelated methods, create multiple smaller interfaces.

```
public interface EngineVehicle {

    void startEngine();

}

public interface Flyable {
 void fly();

}
```

Now truck only implements EngineVehicle, and Airplane can implement both EngineVehicle and Flyable:

```
public class truck implements EngineVehicle {
  @Override
public void startEngine() {
System.out.println("truck engine started.");

    }

public class Airplane implements EngineVehicle, Flyable {
@Override
public void startEngine() {
System.out.println("Airplane engine started.");

    }

    @Override

    public void fly() {

        System.out.println("Airplane is flying!");

    }
}
```

BENEFITS: Each class implements only the behaviors that make sense and the design is cleaner and it avoids forcing irrelevant method implementations.