Nathan Crowley 118429092

****

Theory:

What is Java Abstraction:

- Abstraction is the process of hiding the implementation details from the user of the program. Only the functionality will be provided to the user. This is achieved using abstract classes and interfaces. Abstraction is one of the four major concepts behind object oriented programming.
- Example:

```
//super class
public abstract class Employee {
    public Employee(String name, int paymentPerHour) {
    this.name = name;
    this.paymentPerHour = paymentPerHour;
    }

    public abstract int calculateSalary();
    }
//subclass 1
public class Contractor extends Employee {

  private int workingHours;
  public Contractor(String name, int paymentPerHour, int workingHours) {
     super(name, paymentPerHour);          //inheites the attributes
     this.workingHours = workingHours;
  }
//override the abstract method.
  @Override
  public int calculateSalary() {
     return getPaymentPerHour() * workingHours;
  }
}
```

What is a Java API:

- A way to enable computers to possess a common interface, to allow them to communicate with each other. An API includes classes, interfaces, packages and also their methods, fields and constructors.

****

Rank the principles:

1) **In public classes, use accessor methods, not public fields**
2) **Favour composition over inheritance**
3) **Prefer interfaces to abstract classes**
4) Minimise the accessibility of classes and members

5) Minimize mutability
6) Limit source files to a single top-level class
7) Use interfaces only to define types
8) Prefer class hierarchies to tagged classes
9) Favour static member classes over non-static
10) Design and document for inheritance or else prohibit it
11) Design interfaces for posterity

Report:

1) In public classes, use accessor methods, not public fields

What aspect of good design the principle captures

Accessor methods are javas getters and setters and are used to make **private** data accessible outside of the class. By using these accessor methods, *getters*, as well as mutator methods, *setters*, inside of public classes we can get and set 'sensitive' data from the users of the API. This offers the benefits of **encapsulation.** This helps hide the implementation details from users, giving good abstraction. This is good for design as if a public class was to have public fields without accessor methods it would be very hard to keep control over the code once it has been released to the public. We don't want users to be accidentally setting import data and crashing our code. So the programmer creates a setter which he controls and allows parts of the API to access the data only through getters. Accessor methods are good to preserve the flexibility of classes. Assessor methods can technically be ignored if the class if package-private as having public data wouldn't get very far in a private package. This could lead to cleaner code. But there is always the benefit that code can be manipulated under the control of the programmer at any time using a mutator. Giving good design when it is not necessarily used, cleaner code, and when it is essential to use, more control and flexibility.

If there is a class that is accessible outside its package, it's not package private, then you should provide access methods and mutators. This helps preserve the flexibility of the program as you can call you accessor methods and mutators anywhere where there is sufficient access level.

When deciding if you are to use accessor methods, it is helpful to understand if the data is immutable, objects that don't change, or mutable, can change once it is created. However it is good practice and good design to use accessor methods to avail of the benefits listed above.

Accessor methods also benefit in the fact that there is a single point of contact of attributes, helping with future modifying as you know where to look to modify attributes. Also helps in testing as you can isolate certain methods. Accessor methods help overcoming naming conflicts and name hiding issues in Java, where we give local variables the same name as the attribute. They are useful in having complete control over your program. The programmer can change attributes or edit methods depending on the situation and goal, but it is all under his control as the accessor methods are like roads to the desired attribute.

How it contributes to creating a stable,flexible Java API.

To preserve the **flexibility** of a Java API we use accessor methods as they make it easy to maintain and change code. The API is stable as users cannot interfere with important data as they would need the mutator methods (setters) to be able to do so. Also if a problem

occurs with a class, the programmer can use the accessor methods to retrieve the data and troubleshoot for the problem and even set the data to the correct state.
Another benefit of accessor methods is that data can be made **read-only**(only have an accessor method) *or* **write-only**(only have a mutator method), giving more control to the programmer and giving more flexibility over what code they would like to create.
Accessor methods bring **stability** to objects as they cannot be directly inspected or manipulated. This protects it from corruption by for example, accidental setting of a sensitive variable. With this encapsulation in Java you can hide or restrict the access of critical data in your code ensuring good abstraction.
Accessor methods are good at allowing modification to new and constantly changing rules. If there is requirement to change some attribute then accessors make it very easy as you don't even need to go looking for the original code you can just call `object.getAttribute();` and `object.setAttribute(new_attribute);` giving flexible and easy access to code that could be well hidden in large programs.
Accessor methods are great for encapsulation of validation logic, where we have a requirement to perform some validation before we update or save data. Helpful to place this validation logic in a mutator (setter method) and first check with a conditional statement if the requirement is met and then set the desired attribute to the mutator functions input. Helpful for restricting access to the method.

Example:

```
//employee class with sensitive data "number"
public class Employee {
   private int number;

   //accessor method to return the instances data
   public int getNumber(){
        return this.number;
   }
   //mutator method to set the data as the new given input
   public void setNumber(int new_number){
        this.number = new_number;
   }
}

class testing{
   public static void main(String[] args){
        Employee myWorker = new Employee();
        myWorker.setNumber(10);
        //System.out.println(myWorker.number);       - returns error
        System.out.println(myWorker.getNumber());
   }
}
```

2) **Favour composition over inheritance**

What aspect of good design the principle captures

In java inheritance is a very useful tool, but can lead to fragile code. In java there are two kinds of inheritance, **implementation inheritance** where one class extends another, and **interface inheritance** when a class implements an interface or when dealing with two

interfaces. Problems occur when you are dealing with a concrete class, a class that has an implementation for all its methods, being inherited across packages. These problems do not occur for interface implementation, which I discuss in my first article.
Inheritance violates encapsulation. As a subclass inherits the information from its superclass, encapsulation would state that the information would not change unless it is explicitly altered with a mutator. But a superclass may change and this would result in problems for its subclasses if not accounted for, this will cause the code to crash even without touching the subclass code. This is a direct violation of encapsulation. This is a big drawback of inheritance, as we would constantly have to edit and monitor our subclasses to ensure they are inline with their superclasses.
Inheritance can be troublesome for some as with time as the program develops, there is no guarantee that new features will work with the current inheritance. If there is change in the super class it will alter its subclass too. If a method is deleted in the super class, then it will affect the corresponding subclass. Making things get more complicated as the program size and complexity increases and it may be near impossible to keep an eye out for a small bug somewhere in the code. There is a possibility that the subclass methods might not overwrite anything if the superclass is altered, this will cause the subclass methods to act rogue and have implications that the programmer did not intend.
Another problem of inheritance occurs when we override methods, this can cause fragile code in the subclasses by their superclasses being edited allowing extra functions that were not previously checked by the security systems. Better, but not optimal, to extend a class and add the new methods and forget about older or current methods.

Composition aims to solve some of these problems. Composition is when one class cannot exist without the other .It implements a **has-a** relationship. It is much more flexible.
Composition has many benefits such as allowing to test the implementation of a class independent of the superclass or subclass. This allows us to easily resume code and achieve multiple inheritance. By using composition we can also change some objects at run time. This allows us to dynamically change our programs giving great control and flexibility to the programmer. Composition is achieved by using an instance variable that refers to other objects.
In this example, if a library gets destroyed, then ALL books within this library will also be destroyed. Books cannot exist without a library. This relationship is called composition.
Example:

```
//book class
class Book{
        public String title,author;
        //constructor
        Book(String title, string author){
                This.title = tile;
                this.author = author;
        }
}
//library class
Class Library{
        //reference list of book objects
        Private final List<Book> books;
```

```
        //constructor
        Library(List<Book> books){
            this.books = books;
        }
}
//driver code
Class GFG{
        Public static void main(String[] arg){
            //creating the object of book class
            Book b1 = new Book("Effective Java","Josua Bloch");
            Book b2 = new Book("Thinking in java","Bruce eckle");
            Book b3 = new Book("Java:The complete reference","Herbert schildt");

            //creating list of books
            List<Book> book s= new ArrayList<Book>();
            books.add(b1);
            books.add(b2) ;
            books.add(b3);

            Library library = new Library(books);
            For (Book bk: books){
                System.out.println("Title :" + bk.title + " Author: "+bk.author));
            }
        }
}
```

How it contributes to creating a stable,flexible Java API.

There are many benefits of Composition over inheritance. Such as composition giving the possibility for multiple inheritance, unlike inheritance in Java. If you pursue programming for interface over implementation principle, and you use a type of base class as member variable, you can achieve multiple inheritance indirectly through composition, which allows you to create complex classes using only the characteristics that you need.

Composition also offers **better test-ability** of classes. If you have a compositional relationship between two classes, you can easily build a *test object* This can be done without using superclasses and subclasses, as it is with inheritance.

Composition is great for allowing programmers to reuse code. With one of inheritance's biggest flaws being the fact it breaks encapsulation. Composition does not have this problem because it does not rely on a superclass behavior. Instead you are using a superclass method, making it easier to reuse.

Composition is also good for making code flexible. Using composition ensures you are flexible enough to replace classes with better and more improved versions in the future. All of these add up to make composition more flexible and robust compared to inheritance and making it a good principle for creating API's in java.

Inheritance compared to composition is a **is-a** relationship, for inheritance in java you cannot extend final class, this is not allowed. Also we can only extend, inherit, one class whereas for an interface we can extend many. This leaves inheritance lacking behind interfaces and composition as well.

Inheritance is good for having a superclass and a sub class relationship where you need a subclass to create an object to test your methods in the superclass. It also supports the concept of “reusability”, for example if you wish to create a new class and there is already a class with some data , we do not need to re-write the data we can just reuse the already created attributes. This is very helpful but in terms of abstraction and building stable and flexible API composition is to be preferred.

3) Prefer interfaces to abstract classes

What aspect of good design the principle captures

In java an **interface** is a **completely abstract class**, this is used to group related methods that don't have bodies when they are first created in the interface. The bodies are provided when the interface is **“implemented”** with a subclass. Interfaces cannot be instantiated and have their variables declared as final by default. This subclass will override the interface's abstract methods and give them more context or details. This is beneficial as we can do this to implement multiple inheritance in java, but can only extend one abstract class. It also makes the application **loosely coupled** which can be helpful for better testability, easier to understand code and for scalability. Interfaces also allow for the construction of ‘non hierarchical type frameworks’ which are beneficial for organization of complex data. Interfaces allow for **flexibility** over abstract classes and this is one reason why they are preferred as good API’s should be stable and flexible.

In java an **abstract class** is similar to the interface above but where they differ is that an abstract class may or may not contain abstract methods. Abstract classes cannot be used to create objects, to create objects you first must create a subclass that inherits the abstract class. Abstract methods are methods that, just like in interfaces, do not contain a body in the superclass.

Although you should prefer interfaces to abstract classes for abstraction, abstract classes are useful when you wish to share code among closely related or even package private classes. Also good for when you expect subclasses to have many common methods to the abstract super class. But overall interfacers are better for taking advantage of multiple inheritance and abstraction in Java.

Example:

```
abstract class Animal{
        //abstract method
        public abstract void animalSound();          //no body
        //regular method
        public void sleep(){
                System.out.println(“Zzz”);           //contains body
        }
}
```

Where abstract classes can also contain regular, non-abstract methods in the super class, unlike interfaces.

Example:

```
interface Animal{
```

```
        public void animalSound();
        public void sleep();                    //no body for both
}
```

How it contributes to creating a stable,flexible Java API.

Interfaces ability of multiple inheritance allows for great flexibility, for example if there were two interfaces that were similar, a *singer interface* and a *songwriter interface*, a class could fall into both categories and with interfaces this is allowed.

```
public interface SingerSongwritter extends Signer, Songwriter{}
```

Interfaces also use **wrapper class**, a useful way to use primitive data types (int,boolean,etc) as objects, these give broader scope to a programmer to allow them to use primitive data types in situations where they would usually not be allowed by the compiler. Without these, the programmer would have to resort to inheritance which results in less optimal classes. So with interfaces you can ensure that you follow programming to interface compared to implementation patterns or abstract classes. This will lead to more flexibility in your program which is beneficial to APIs.

However interfaces are to be preferred over abstract classes, the most optimal outcome would be to combine both approaches using an **abstract skeletal implementation class** alongside an interface. Where the interface allows you to define types and use default methods, which are good for adding new functionality to existing interfaces and to ensure compatibility with older code versions, while using the abstract skeletal implementation class to implement the remaining interface methods. The abstract skeletal implementation class can be implemented to make it easier for programmers to access the benefits of implementing an interface.

The advantages of skeletal implementation is that they give the benefits of abstract classes with the constraints usually found with using these classes. This kind of implementation is relatively simple to write, concentrating on the interface and deciding which methods are the primitives and this will be used later on as the abstract methods.

Resources:

- Bloch 2018, Effective Java, Third Edition, Joshua Bloch Addison-Wesley, 2018
- https://javapapers.com/core-java/abstract-and-interface-core-java-2/difference-between-a-java-interface-and-a-java-abstract-class/#:~:text=Main%20difference%20is%20methods%20of,that%20implements%20a%20default%20behavior.&text=An%20abstract%20class%20may%20contain,interface%20are%20public%20by%20default.
- https://www.w3schools.com/java/java_interface.asp
- https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html
- https://en.wikipedia.org/wiki/Loose_coupling
- https://javapapers.com/core-java/abstract-and-interface-core-java-2/difference-between-a-java-interface-and-a-java-abstract-class/#:~:text=Main%20difference%20is%20methods%20of,that%20implements%20a%20default%20behavior.&text=An%20abstract%20class%20may%20contain,interface%20are%20public%20by%20default.
- https://yourbasic.org/algorithms/your-basic-api/
- https://www.programiz.com/java-programming/wrapper

- https://www.c-sharpcorner.com/UploadFile/3614a6/accessors-and-mutators-in-java/#:~:text=In%20Java%20accessors%20are%20used,are%20also%20known%20as%20setters.
- https://www.geeksforgeeks.org/concrete-class-in-java/
- https://www.geeksforgeeks.org/composition-in-java/#:~:text=The%20composition%20is%20a%20design,that%20refers%20to%20other%20objects.
- https://javarevisited.blogspot.com/2013/06/why-favor-composition-over-inheritance-java-oops-design.html#axzz6eOdoe1NB
- https://javarevisited.blogspot.com/2013/06/why-favor-composition-over-inheritance-java-oops-design.html#axzz6eOdoe1NB
- https://docs.oracle.com/javase/tutorial/java/IandI/abstract.html