CSCI 4448 - OOAD Project 1

Students:

- Lucas Gama (student id: 108696592)
- Felipe Lima (student id: 109290055)

Abstraction: This concept exists to hide unnecessary information from the users giving a large pool of data. The idea is that, with abstractions, we only show to users relevant information and details from a certain object. Abstraction can be useful because instead of understanding how each method in your code is implemented, you just need to know which methods are available to use and what their parameters are. This concept is considered one of the most important concepts in object-oriented programming as it helps to reduce "programming complexity". Abstraction can be applied in classes using the keyword *abstract* followed by its class name or as methods (also using the keyword *abstract* in front of the method). Abstract classes cannot create objects or be instantiated, but can be inherited.

Here is an example of a abstract method:

```
abstract void moveTo (double deltaX, double deltaY);
```

and and example of an abstract class:

```
class abstract public ClassExample{
    //declare its atributes here
    //declare all necessary methods here
}
```

- Source: Link 1, Link 2, Link 3, Link 4

Encapsulation: This is a process that allows us to operate data into a single entity by "wrapping" the data and the code. Encapsulation exists to make a system easier for an end user to handle it and it can be interpreted as a "protective wrapper that stops random access of code defined outside that wrapper". In relation to classes, encapsulation can be used to control the access of methods and attributes from a certain class. It will guarantee the protection of a class' data as well as define where that class can be manipulated. To use encapsulation, we can use the keyword *private* followed by the variable name. Lastly, in order to get access to encapsulated variables, we can use SET (to set a value) and GET (to retrieve a value).
Here is an example of a class that uses encapsulation.

Source: Link 1

Polymorphism: In object oriented programming, polymorphism is the capability of an object to act in different ways depending on the manner in which it was instantiated. The advantage of polymorphism is to reduce lines of codes because you no longer have to write different classes to perform an action. Polymorphism will select the right method to be used based on the class the object is derived from.

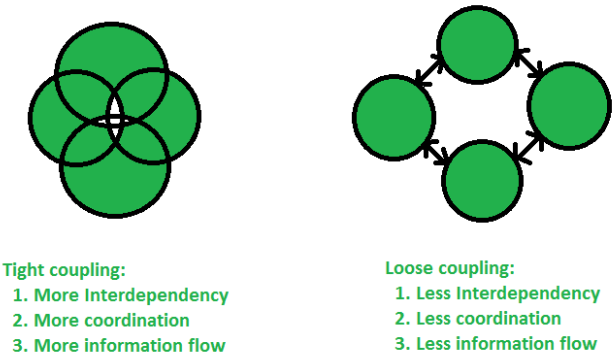
On lines 24 and 26 we can see an example of polymorphism:

```
6  abstract class Mammal {
7    public abstract double amountToFeedKG();
8  }
9
10 class dog extends Mammal {
11   public double amountToFeedKG(){
12     return 0.8;
13   }
14 }
15
16 class lion extends Mammal {
17   public double amountToFeedKG() {
18     return 20.0;
19   }
20 }
21
22 class FeedControl {
23   public static void main(String args[]){
24     Mammal mammal1 = new dog();
25     System.out.println("The amount to feed a dog in KG is: " + mammal1.amountToFeedKG());
26     Mammal mammal2 = new lion();
27     System.out.println("The amount to feed a lion in KG is: " + mammal2.amountToFeedKG());
28   }
29 }
```

- Source: Link 1

Coupling: In object oriented programming, coupling is the strength of a connection between two modules. In simpler words, it is how much one module is going to be affected by a change in another. In the notion of a class, loose coupling means a connection in which one module communicates with another module through a simple and stable interface and the internal implementation of the other module does not have great effect or knowledge of the prior. While tight coupling describes the opposite, meaning, a change in one module usually has a ripple effect of changes in other modules.

```java
class Size
{
    public static void main(String args[])
    {
        Painting b = new Painting(5,5);
        System.out.println(b.volume);
    }
}
class Painting
{
    public int size;
    Painting(int width, int height)
    {
        this.size = width * height;
    }
```



Tight coupling ↑↑

- Source: Link 1, Link 2

Cohesion: In object oriented programming, cohesion is how closely related the actions of a class are. In other words, it is how clear the responsibility of each method is. High cohesiveness is related to easy maintenance of a class and it makes it more reliable. A class or method that performs one specific action or has one clearly defined purpose is strongly cohesive or has high cohesion. A class with low cohesion is usually messy, hard to understand, difficult to maintain and reuse.
Example of a cohesive program:

```java
class Add {
    int a = 5;
    int b = 5;
    public int add(int a, int b)
    {
        this.a = a;
        this.b = b;
        return a + b;
    }
}

class Display {
    public static void main(String[] args)
    {
        Add a = new Add();
        System.out.println(a.add(5, 5));
    }
}
```

- Source: Link 1, Link 2

Identity: Identity is the property that is responsible for distinguishing one object from all the others. A unique identifier is responsible for setting apart every object created in a class. It can be used to compare two variables and check if they point to the same address or object. An object should have an existence independent of its value. Classes should only be created for objects that have a unique identity in the domain.

```
Class item1;
// creates an instance (object) of a class. No matter what values this takes
//on over the course of its lifetime, its name will always be the same object
Class * item2 = new Class(Point(75, 75));
// Allocates a pointer
// The object doesn't exist until run-time.
```

- Source: Link 1, Link 2