## Encapsulation:

Differences between OOP

Python vs Java

# Encapsulation:

# Public, Private and Protected & Getters and Setters

Submitted by Alan John Heslop

Date: 11/10/2020

Officially, within Python, there is no such thing as “private” variables, however, there are methods to using, the term loosely, protected, and private instances. However, the user can set specific instance an attribute such as:

-        Default variable:

o   Self.foo denotes this is a public variable.

-        Single leading underscore:

o   Self.\_foo denotes that this is a protected instance variable.

-        Double leading underscore:

o   Self.\_\_foo denotes that this is a “private” instance variable.

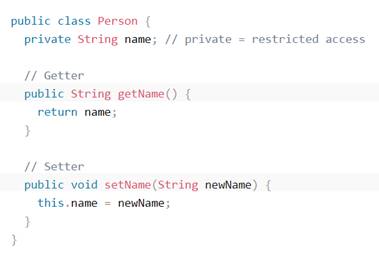
               Within Java encapsulation can be triggered when the user chooses a variable method such as Public, Private, or Protected. A public method can be called by any object and can be set as a class or inside the class. A private method is the most used variable because these variables are only visible to the class in which they belong, and If not implemented correctly this could lead to private data being accessed easily. Lastly, we have another method known as protected and this method is only available to those within all classes in the same package, and within subclasses in other packages. As well as having dedicated variables to protect and privatize strings of data, you can also customize the methods as well.

## Getters and setters

A getter and setter are methods that have different uses on a variable. A getter is a method that can be used to read the value(s) of a variable, and a setter is a method that can be used to update a variable.

Below we can see that the “Getter” and “Setter” are clearly defined for the user of calling/setting the string to update the information or getting the string to read the information -in this case ‘name’.

Example:



(image from: <https://www.w3schools.com/java/java_encapsulation.asp> – 11/10/2020)

However, in a pythonic method, this is not available for classes and as such, we need to revert to a dynamic method known as a property decorator. The property decorator allows us to define a method that we can access to either retrieve or update data.

Example:

class person:

    def \_\_init\_\_(self):

        self.\_\_name=''

    @property

    def name(self):

        return self.\_\_name

    @name.setter

    def name(self, value):

        self.\_\_name=value

The above shows that you can create a person class and set the methods the same as before, however you can then call the “setter” method to p = person() then p.name=”bob”. Encapsulation is also available to use on the property method, as above it shows the name is a private variable.

Graphical user interface, text

Description automatically generated