https://doi.org/10.26682/csjuod.2023.26.2.35
Journal of University of Duhok., Vol. 26, No.2 (Pure and Engineering Sciences), Pp 379-389, 2023
4th International Conference on Recent Innovations in Engineering (ICRIE 2023) (Special issue)

COMPARATIVE ANALYSIS: EXPLORING ENCAPSULATION IN JAVA AND PYTHON PROGRAMMING LANGUAGES

SKALA KAMARAN OMER*, STAR DAWOOD MIRKHAN*, NYAN NAJAT HUSSEIN*, AVEEN ZUBER ALI*, TARIK AHMED RASHID *, HUSSEIN MOHAMMED ALI**, MAHMOOD YASHAR HAMZA** and POORNIMA NEDUNCHEZHIAN***

*Computer Science and Engineering, University of Kurdistan Hewler, Kurdistan Region- Iraq
**Computer Engineering Department, Tishk International University, Kurdistan Region, Iraq
***School of Computer Science and Engineering, Vellore Institute of Technology, Vellore- India

(Accepted for Publication: November 27, 2023)

ABSTRACT

Encapsulation is a fundamental principle of object-oriented programming, which allows for the hiding of implementation details and the protection of data from unauthorized access. This paper evaluates the concept of encapsulation in the programming languages Java and Python. The study examines the ways in which encapsulation is implemented in each language, including access modifiers and methods of data hiding. This research paper's findings demonstrate that while both languages support encapsulation, the syntax and specific implementation vary in each of them. In particular, Java's use of access modifiers such as private, protected and public allows for a stricter level of encapsulation compared to Python's use of the '_' and '__' prefix to denote private variables and methods.

KEYWORDS: Encapsulation; Java; Python; Object-oriented programming; Security

1. INTRODUCTION

Encapsulation plays a role, in OOP by serving as a shield that prevents programs from directly accessing data. It offers advantages, such, as isolating data, improving flexibility, promoting reusability, and ease of testing. Access modifiers are used to encapsulate data. Encapsulation allows a structured data object's values or state to be held privately within a class, preventing unauthorized users from having direct access to it. As one of the principles of OOP languages, which offers encapsulation, the technique of implementation can differ. In this article, the Java and Python programming languages will be compared in terms of providing encapsulation. To accomplish encapsulation in Java language, a class must be declared as private, and both public (Setter and Getter) methods must be used to access and update the values of private variables in the private class. However, this is not the case in Python. Python and Java vary in terms of simplicity for both (Setter and Getter) methods, debugging and testing efficiency, and the number of standard libraries, but both languages may match in terms of security, readability, and many other criteria that will be covered in this article. In this study, the utilization of encapsulation is done by using Java version 17 and Python version 3.11 to investigate the implementation of

skala.kamaran@ukh.edu.krd
satar.dawood@ukh.edu.krd,
nyan.najat@ukh.edu.krd,
aveen.zuber@ukh.edu.krd
tarik.ahmed@ukh.edu.krd
hussein.mohammed@tiu.edu.iq
mahmood.yashar@tiu.edu.iq

encapsulation in both languages. The primary objective of this research is to compare both Java and Python languages by highlighting their differences in encapsulation methods. The structure of the paper follows as: background of encapsulation for both languages is presented in Section 2, the code implementations of encapsulation for both Java and Python, as well as advantages and disadvantages are represented in Section 3, and the conclusion of this paper is briefly presented in Section 4.

2. BACKGROUND

2.1 History of Encapsulation in Java and Python

Even though various languages might be useful for programmers, this article will describe and compare two languages with the intention of simplicity. Java and Python are well-known and highly rated programming languages on reliable websites. They are popular, and there is a high demand for them in the job market. A beginner's programming language should include characteristics such as credibility, simplicity, accessibility, and ease of learning. Encapsulation is the process of enclosing data in a single class. It is the technique that links the code to the data that it manipulates (Khoirom, Sonia, Laikhuram, Laishram, and Singh, 2020). Encapsulation is also a barrier that prevents programs from outside the barrier from accessing the data. Variables or data in a class are concealed from all other classes and may only be accessible through public methods of the class in which they were defined (Ma and Foster, 2007). James Gosling, Mike Sheridan, and Patrick Naughton created the Java programming language in June 1991 at Sun Microsystems, which was a multinational company known for its computer hardware, software, and network technologies, released it as a significant product in 1995 in which it was renamed from Oak to Java. Many of the popular tools and applications that operate at Sun Microsystems are powered by Java (Johnson and Chandran, 2021).

The concept of encapsulation is used in OOP to create abstract datatypes that restrict modifications to their external interface. The outcome is the development of programs that possess enhanced maintainability and troubleshooting capabilities. The use of different access modifiers, such as private or public, when declaring variables could aid the management of encapsulation to a certain degree. Nevertheless, it is important to note that in programming languages such as Java, only the names of the variables are protected, but the actual objects being referred to by these variables are not secured. It is possible for an external object to get references to objects stored in the private variables of a compound object via method calls (Skoglund, 2003).

Encapsulation is a fundamental principle in the Java programming language since it controls the accessibility of data. It serves many purposes, including:

- Adapting the code to meet specific needs.
- Facilitating loose coupling between components.
- Simplifying the application to effectively accomplish its objectives.
- Permits the modification of a code section without affecting other program features or lines of code.

In the late 1980s, Guido van Rossum created Python at the Central Wiskunde & Informatica (CWI) in the Netherlands (Rossum & Guido, 2007), where it was implemented in December 1989. Python has been proposed as an alternative to the ABC programming language that can handle exceptions and interact with the Amoeba operating system. Python could be a good choice for both learning and global programming. Python is a high-level, OOP language. In recent years, Python has earned a reputation as an extremely user- and beginner-friendly programming language (Johnson & Chandran, 2021). The primary benefit of implementing encapsulation in Python is that users do not need

to understand the method and data design and can instead concentrate on utilizing these useful, encapsulated entities within their applications. Consequently, the code becomes more structured and well-arranged. Additionally, the user experience has been greatly improved, making applications easier to understand. Encapsulation serves as a safeguard, against alterations or removals. It offers the advantage of protecting data and methods, from changes. In Python encapsulation is achieved through access modifiers. These modifiers provide users with a level of security by ensuring that access conditions are not violated.

2.2 Previous Related Works

In the world of OOP, developers have long recognized the importance of encapsulation and information hiding. These principles have the potential to enhance software development improve software quality and make maintenance easier. This section presents an overview of studies that delve into the implementation and effectiveness of encapsulation and information hiding in OOP languages. It offers insights, into the difficulties and subtleties associated with these principles.

2.2.1 Guidelines and Best Practices

Many coding conventions and guidelines stress the significance of encapsulation and concealing information. For example, the "Java Programming Language Code Conventions" which is publicly available in their official website, advise, against declaring instance or class variables as public unless there is a justification. The purpose of these guidelines is to encourage programmers to be disciplined and enhance code quality by advocating coding practices.

2.2.2 Empirical Studies

Numerous real-world software development studies have delved into the implementation and adherence, to encapsulation and information-hiding principles. (Menzies & Haynes, 1996) analyzed Smalltalk applications to assess the information hiding's prevalence. They posited that if programs adhere to the principle of encapsulation, there should be fewer calls to other classes. However, their examination of 2,000 classes across five applications revealed a relatively low incidence of information concealment. (Elish, Omar, & Offutt, 2002) delved into the adherence to coding standards and practices within open-source Java classes. They reported that prohibiting the creation of public member variables was the third most problematic coding technique, suggesting that maintaining encapsulation can be challenging, even for experienced programmers. (Fleury & E, 2001) conducted a study involving 28 students studying OOP. Interestingly, many students prioritized reducing the number of classes and lines of code over encapsulation. This perspective underscores the need for improved education and awareness regarding encapsulation's benefits and principles.

2.2.3 Challenges and Critiques

While encapsulation and information hiding are lauded in theory, their practical application can be complex and subject to various challenges:

- **Language Support:** Some programming languages may not provide robust support for encapsulation and information hiding. For example, C++ cannot separate the declaration of public and private member variables, potentially hindering the practice of encapsulation.

- **Inheritance Implications:** The inheritance model in certain languages, such as Smalltalk, can compromise encapsulation by granting subclasses full access to superclass member variables. This can make it challenging to update superclass code without impacting clients.

2.2.4 Impact on Software Quality

Despite the empirical evidence suggesting that encapsulation and information hiding are not universally observed, their precise impact on

software quality remains an open question. Few studies have rigorously examined whether adherence to these principles leads to higher-quality software or whether deviations have noticeable consequences.

This research article (Pizarro-Vasquez, Barahona, & Mig, 2021), has shown that the software quality has been improved when encapsulation is implemented, also, for the runtime speed, the output of the system is faster and more robust, which is presented in their results, in many research articles, such as (Arvanitou, Ampatzoglou, Chatzigeorgiou, & Carver, 2020). In terms of debugging, encapsulation has shown remarkable results in several ways, including isolation of the code, which makes it easier to check and debug for the programmers, also, in terms of testing, especially unit testing, have an effective approach for error handling. In addition, encapsulation can be used for debugging to maximize code robustness (Arvanitou, Ampatzoglou, Chatzigeorgiou, & Carver, 2020).

Table (1 (Nilsen, 2007), represents an in-depth evaluation of how encapsulation is carried out in both Java and Python programming languages, with a particular focus on the differences between these two languages. It describes important things including access modifiers, the benefit of the Getter and Setter methods, the amount of difference implementation, and the creation of properties, and gives a brief example for each language. It also addresses how encapsulation affects inheritance and how both Java and Python handle the appearance of classes across class hierarchies (Nilsen, 2007).

Table (1) :- Comparison of encapsulation in Java and Python.

Aspect	Java	Python
Access Modifiers	Public, Private, Protected, Default	Public, Private
(Getter and Setter) Functions	Common practice	Less common, direct attribute access
Encapsulation Enforcement	Strong (Compiler enforces access)	Weak (interpreter)
Properties	JavaBeans for creating properties	property decorator
Example	private int age Getter and setter methods	property and attribute setter functions
Inheritance Impact	Subclasses access protected members	Subclasses access private members

Table(2 (Khoirom, Sonia, Laikhuram, Laishram, & Singh, 2020), provides a summary of the benefits and drawbacks of encapsulation, in Java. It outlines the advantages, such as improved data security, precise control over attribute access, flexibility, easy maintenance, and simplified debugging. At the time it acknowledges the disadvantages like verbosity caused by Getter and Setter methods, performance overhead lack of flexibility, and the potential for excessive code duplication, due to boilerplate code (Khoirom, Sonia, Laikhuram, Laishram, & Singh, 2020).

Table(2) :- Advantages and disadvantages of Java encapsulation.

Advantages	Disadvantages
Improved data security and integrity	More code due to (Getter and Setter) methods
Have Control access to private classes	Performance overhead
Easily can adapt and modify the code	Not flexible in some designs
Enhanced maintainability	This can lead to a lot of repetitive code
Enables debugging and error tracing	There may be complicated inheritance hierarchies.

Table (3 (Khoirom, Sonia, Laikhuram, Laishram, & Singh, 2020), illustrates the summary which outlines the advantages and disadvantages of encapsulation, in Python language. It highlights the benefits, such as the simplicity of using properties the reduction in code, flexibility, dynamic attribute behavior improved code readability and conciseness. Moreover, it also discusses the drawbacks like the lack of access control (no 'private' keyword) potential misuse, stringent encapsulation compared to Java decreased protection for internal attributes and added complexity when handling inheritance without the 'protected' access modifier. These tables provide insights into understanding how encapsulation works in both Java and Python. They also allow for a comparison between their approaches to this concept, in OOP (Khoirom, Sonia, Laikhuram, Laishram, & Singh, 2020).

Table (3) :- Advantages and disadvantages of Python encapsulation.

Advantages	Disadvantages
Syntax made easier with properties	Access control is limited (no "private").
Reduce the repetitive code	Possible abuse (straight access to attributes)
Class characteristics that can be changed easily	Separation is not as precise as it is in Java.
Supports the behavior of flexible attributes	Less safety for objects on the inside
Made the code easier to read and simpler.	There is no "protected" word when it comes to inheritance.

3. Implementation of Encapsulation in Java and Python

This section will explore the practical considerations involved in establishing encapsulation in the Java and Python programming languages. Both languages have unique methods for accomplishing encapsulation, and it will thoroughly examine these features of the encapsulations.

3.1 Encapsulation in Java Language

In Java, the concept of encapsulation is enforced by using access modifiers and Getter and Setter methods. These access modifiers, such, as public, protected, and default (package private) provide developers with the ability to manage the visibility of class members. By declaring attributes as private and offering Getter and Setter methods Java ensures that data encapsulation remains intact. This approach enhances security and control over data access, which results in code integrity, due to the necessity of using Getter and Setter methods (Khoirom, Sonia, Laikhuram, Laishram, & Singh, 2020).

3.1.1 Advantages of Encapsulation in Java

```java
public class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Getter method for name
    public String getName() {
        return name;
    }

    // Setter method for age
    public void setAge(int age) {
        if (age >= 0) {
            this.age = age;
        }
    }
}
```

Fig. (1) :- Improved Control.

This Java example in Fig. (1, highlights the benefits of encapsulation by making a Person class with the name and age as private properties. The class allows you to access and change these characteristics in a specific manner. The get Name method enables outside code to be able to get the name attribute and guarantee that it only allows for reading. The set Age method gives controlled access to the age attribute, ensuring that it cannot be set to a negative number. This separation makes it easier to control who can access and alter the data, therefore the changes that are not meant or not valid will not take place (Khoirom, Sonia, Laikhuram, Laishram, & Singh, 2020)

3.1.2 Disadvantages of Encapsulation in Java

The Java example below Fig. (2, demonstrates a potential issue with encapsulation, which is that it can make the code lengthy. The student class has private variables for name and age while offering each field its own Getter and Setter methods. Encapsulation helps keep control over how data is obtained, but it makes code lengthier because each property needs a Getter and a Setter method. This can make the code longer and harder to understand, especially for classes with a lot of properties (Khoirom, Sonia, Laikhuram, Laishram, & Singh, 2020)

skala.kamaran@ukh.edu.krd
satar.dawood@ukh.edu.krd,
nyan.najat@ukh.edu.krd,
aveen.zuber@ukh.edu.krd
tarik.ahmed@ukh.edu.krd
hussein.mohammed@tiu.edu.iq
mahmood.yashar@tiu.edu.iq

```java
public class Student {
    private String name;
    private int age;

    public Student(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Getter method for name
    public String getName() {
        return name;
    }

    // Setter method for name
    public void setName(String name) {
        this.name = name;
    }

    // Getter method for age
    public int getAge() {
        return age;
    }

    // Setter method for age
    public void setAge(int age) {
        if (age >= 0) { this.age = age; }
    }
}
```

Fig. (2) :- Increased Code Length.

3.2 Encapsulation in Python language

In Python encapsulation is approached in a manner with conventions being relied upon rather than strict enforcement. Though Python doesn't have members it utilizes a single leading underscore (e.g., _variable) to indicate that an attribute is intended for internal use. Additionally, developers can make use of the @property decorator to define Getter methods. If required @<attribute>. setter methods to manage attribute access. This approach reduces code and simplifies attribute access, but it depends on convention and may not be as strict as Java's enforcement (Khoirom, Sonia, Laikhuram, Laishram, & Singh, 2020).

3.2.1 Advantages of Encapsulation in Python

The Python code example below illustrates in Fig. (3, the Employee class with the private variables with two properties including (_name and _salary) which represents how prevention can be useful for data privacy. Even though that Python is not as strict about access rules as Java, the use of a single underscore in front of a property of the variables indicates that it is private and should not be viewed through outside code. The class also has a set salary method that makes it impossible to change the salary negatively. This separation makes the code easier to read and protects it from people who shouldn't be able to get to it (Khoirom, Sonia, Laikhuram, Laishram, & Singh, 2020).

skala.kamaran@ukh.edu.krd
satar.dawood@ukh.edu.krd,
nyan.najat@ukh.edu.krd,
aveen.zuber@ukh.edu.krd
tarik.ahmed@ukh.edu.krd
hussein.mohammed@tiu.edu.iq
mahmood.yashar@tiu.edu.iq

```python
class Employee:
    def __init__(self, name, salary):
        self._name = name      # Protected attribute
        self._salary = salary  # Protected attribute

    # Getter method for name
    def get_name(self):
        return self._name

    # Setter method for salary
    def set_salary(self, salary):
        if salary >= 0:
            self._salary = salary
```

Fig. (3):- Protection from Unauthorized Access.

3.2.2 Disadvantages of Encapsulation in Python

The Python code below in Fig. (4, demonstrates a possible downside of encapsulation which is the requirement for an additional code. The student class, which has the private properties __name and __age. Separate Getter and Setter methods are required to access or alter these properties. While encapsulation protects attributes from direct outside accessibility, it requires additional code to build and preserve these methods, increasing code length and possibly making the source code more complicated as compared with languages that allow direct attribute access (Khoirom, Sonia, Laikhuram, Laishram, & Singh, 2020).

```python
class Student:
    def __init__(self, name, age):
        self.__name = name  # Private attribute
        self.__age = age    # Private attribute

    # Getter method for name
    def get_name(self):
        return self.__name

    # Setter method for name
    def set_name(self, name):
        self.__name = name

    # Getter method for age
    def get_age(self):
        return self.__age

    # Setter method for age
    def set_age(self, age):
        if age >= 0:
            self.__age = age
```

Fig. (4) :- Additional Coding Required.

skala.kamaran@ukh.edu.krd
satar.dawood@ukh.edu.krd,
nyan.najat@ukh.edu.krd,
aveen.zuber@ukh.edu.krd
tarik.ahmed@ukh.edu.krd
hussein.mohammed@tiu.edu.iq
mahmood.yashar@tiu.edu.iq

Table (4 examines the context of OOP, in which encapsulation is a mechanism that manages the accessibility of a class's properties and methods. Access to private variables is restricted to the class inside which they are declared. Subclasses are not permitted access to private variables. Access to protected members is restricted to the same class and its subclasses and cannot be directly obtained via object instances. In contrast, public members may be accessed from any location, including the class itself, its subclasses, and via instances of the class. The below table presents a comprehensive overview of the primary accessibility attributes associated with various forms of encapsulation (Boyarsky, Jeanne, & Selikoff, 2020).

Table (4):- Accessibility states of encapsulation.

Type	Use your class to access	Use sub-class to access	Use objects to access
Private	Yes	No	No
Protect	Yes	Yes	No
Public	Yes	Yes	Yes

Table (5 presents the encapsulation syntax in both languages, which is a key principle in Python and Java programming languages and serves as a mechanism for regulating access to class members, including both attributes and methods. In the Python programming language, it is customary to represent private variables by using a single underscore as a prefix (e.g., _VariableName). On the other hand, protected variables are not expressly specified but are indicated by the same underscore prefix. Nevertheless, these norms do not constitute stringent access control measures, rather, they depend on the conscientiousness of developers. In the Java programming language, however, access control is implemented via explicit keywords. Private variables are declared using the private keyword, such as private String VariableName. Similarly, protected variables are stated using the protected keyword, for example, protected String VariableName. Public variables are declared using the public keyword, as seen by the public String VariableName. The following table is a concise overview of the syntax used for encapsulation in the programming languages Python and Java (Khoirom, Sonia, Laikhuram, Laishram, & Singh, 2020).

Table (5) :- Defining Syntax of Encapsulation in Python and Java.

Type	Python	Java
Private	___Varible-Name	private String VaribleName
Protect	_Varible-Name	protect String VaribleName
Public	Varible-Name	public String VaribleName

3.3 The difference between Setter and Getter methods for both Python and Java

In OOP, Getters and Setters methods are used because of encapsulation, which is presented with

code examples in Fig. (5, for both languages. This maintains the privacy of private variables to make sure that only the Getters and Setters methods may access and modify the variables. The Java language should create two methods with different names to implement Getters and Setters but in Python, it can put the same name for Getters and Setters but with different attributes and properties (Farooq & Khan, 2023).

```python
class MyClass:
    def _init_(self):
        self._variable = None
        # Private variable with
        # an underscore prefix

    def get_variable(self):
        return self._variable

    def set_variable(self, value):
        self._variable = value

# Usage
obj = MyClass()
obj.set_variable(42)
print(obj.get_variable())  # Output: 42
```

```java
public class MyClass {
    private int variable;
    // Private variable

    // Getter method
    public int getVariable() {
        return variable;
    }

    // Setter method
    public void setVariable(int value) {
        this.variable = value;
    }
}

// Usage
MyClass obj = new MyClass();
obj.setVariable(42);
int value = obj.getVariable();
// value will be 42
```

Fig. (5) :- Getters and Setters Methods Example for Python and Java.

4. CONCLUSION

This paper examines the notion of encapsulation in both Java and Python, underlining its importance in protecting data integrity, simplifying code maintainability, and regulating access to class attributes. By using examples, we have shown the implementation of encapsulation and its associated merits and drawbacks in several codes. Nevertheless, the domain of software engineering exhibits a dynamic nature, whereby the advancements in technology give conception to novel encapsulation prospects and obstacles. Potential areas for further investigation include the exploration of methods to enhance the efficiency of encapsulation strategies. This might include the examination of automated code generation tools that mitigate the length often associated with encapsulation in the Python programming language. Additionally, alternative mechanisms in Java could be explored to achieve a balance between control and code conciseness. In addition, with the growing complexity and distribution of software systems, there is a demand for study to explore the application of encapsulation concepts inside distributed and microservices architectures. Furthermore, it is necessary to do additional research on the effects of encapsulation on performance and memory use in contexts with limited resources. The notion of encapsulation is an important aspect in the field of software development since continuous research will significantly contribute to its implementation in future software systems.

REFERENCES

Arvanitou, E.-M., Ampatzoglou, A., Chatzigeorgiou, A., &
Carver, J. C. (2020). Software Engineering Practices

for Scientific Software. *Journal of Systems and Software*. doi:https://doi.org/10.1016/j.jss.2020.110848

Boyarsky, Jeanne, & Selikoff, S. (2020). Methods and Encapsulation. In J. Boyarsky, & S. Selikoff, *OCP Oracle Certified Professional Java SE 11 Programmer I Study Guide: Exam 1Z0-815* (pp. 247-295). Wiley. doi:10.1002/9781119584773

Elish, Omar, M., & Offutt, J. (2002). The adherence of open source Java programmers to standard coding practices. 193-198.

Farooq, M. S., & Khan, T. z. (2023). Comparative Analysis of Widely use Object-Oriented Languages. *arXiv e-prints*.

Fleury, & E, A. (2001). Encapsualtion and reuse as viewed by java students. *ACM SIGCSE Bulletin, 33*(1), 189-193.

Johnson, B., & Chandran, A. S. (2021). COMPARISON BETWEEN PYTHON, JAVA AND R PROGRMMING LANGUAGE IN MACHINE LEARNING. *International Research Journal of Modernization in Engineering Technology and Science*, 3288-3293.

Khoirom, S., Sonia, M., Laikhuram, B., Laishram, J., & Singh, T. D. (2020). Comparative Analysis of Python and Java for Beginners. *International Research Journal of Engineering and Technology (IRJET), 07*(08), 4384-4407.

Ma, K.-K., & Foster, J. S. (2007). Inferring Aliasing and Encapsulation Properties for Java. *OOPSLA'07,*, 1-18.

Menzies, T., & Haynes, P. (1996). Empirical Observations of Class-level Encapsulation and Inheritance. *Technical report, Department of Software Development, Monash University*, 1-8.

Nilsen, K. (2007). Improving abstraction, encapsulation, and performance within mixed-mode real-time Java applications. *Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems*, 13-22.

Pizarro-Vasquez, G. O., Barahona, F., & Mig. (2021). Encapsulation Component and Its Incidence into Scientific Software Performance. In *Smart Innovation, Systems and Technologies.* Springer. doi:https://doi.org/10.1007/978-981-16-4126-8

Rossum, V., & Guido. (2007). Python Programming Language. *USENIX annual technical conference*.

Skoglund, M. (2003). Practical Use of Encapsulation in Object-Oriented Programming. *Software Engineering Research and Practice* , 554-560.