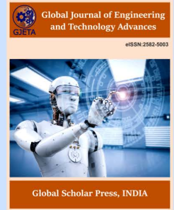
Global Journal of Engineering and Technology Advances

eISSN: 2582-5003

Cross Ref DOI: 10.30574/gjeta

Journal homepage: https://gjeta.com/



(Research Article)

Studies in object-oriented programming backbone implementations

Monday Eze [1,*], Charles Okunbor [2] and Umoke Chukwudum [3]

[1] *Department of Computer Science, Babcock University, Nigeria.*
[2] *Department of Computer Science, Admiralty University, Delta State, Nigeria.*
[3] *Department of Vocational and Technical Education, Alex-Ekwueme Federal University, Nigeria.*

Global Journal of Engineering and Technology Advances, 2021, 08(03), 020–031

Publication history: Received on 20 July 2021; revised on 24 August 2021; accepted on 26 August 2021

Article DOI: https://doi.org/10.30574/gjeta.2021.8.3.0119

Abstract

This work is a combination of conceptual and hands on based study aimed at laying a foundation for practical Object-Oriented software construction. First it presents a conceptual study of a number of backbone concepts of modern Object-Oriented Programming (OOP) languages. Secondly, it attempts to demonstrate real-life implementations of these concepts using Python Programming Language. This work touches on practical issues on Class and Object Creation, especially on the syntax and creation, and demystifies the subject matter using a simple table of rules. The OOP concept of Inheritance was studied, with focus on the three major types of inheritance. The self-argument, and constructors were studied, with focus on the three constructors - default, parameterized, and non-parameterized constructors. A brief discussion, and pictorial illustration was also made on the disparity between normal mathematical functions and OOP method calls. Further areas of studies are the concept of overriding between the parent and child class, as well as the OOP puzzle commonly known as Diamond Problem, including code segment and diagrammatic illustration of Python-based solutions. There are a number of other back-bone concepts in OOP not covered in this study, such as Encapsulation, Abstraction, Meta-Programming, among others, which will form areas of focus in future studies. Effort was made to enhance the overall presentation through practical illustrations using source codes, annotated diagrams, and discussions. It is hoped that this work will be very useful to researchers and other practitioners in Object Oriented implementations.

Keywords: Object-Oriented Programming; Constructors; Class; Inheritance; Diamond Problem

1. Introduction

Though Object Oriented Programming (OOP) [1] is considered to be a modern programming paradigm, the root goes back to 1967 when the first OOP language known as Simula 67 [2] was introduced. Understanding the relative strength of different programming languages in handling problems in different domains [3] is key. Different programming languages have their individual areas of strength and focus. In other words, their usage may vary drastically, depending on the type of problems being solved. A study of survey in programming languages [4] helps researchers to understand the strengths and weaknesses of different languages, and the choice of which paradigm fits which problems. Some of the major programming paradigms [5] are shown in Fig. 1. As shown in the figure, some of the major programming paradigms are imperative [6], object oriented [7], concurrency [8], functional [9], logic and event-driven [10] programming. This research focuses on Object Oriented Programming with the practical implementations based on Python 3.8. The choice of Python in this research is based on a number of its strengths and attributes. One of such attributes is the fact that Python is a multi-paradigm [11] programming language. This means, that it supports different programming approaches.

* Corresponding author: Monday Eze
Department of Computer Science, Babcock University, Nigeria.

Copyright © 2021 Author(s) retain the copyright of this article. This article is published under the terms of the Creative Commons Attribution Liscense 4.0.

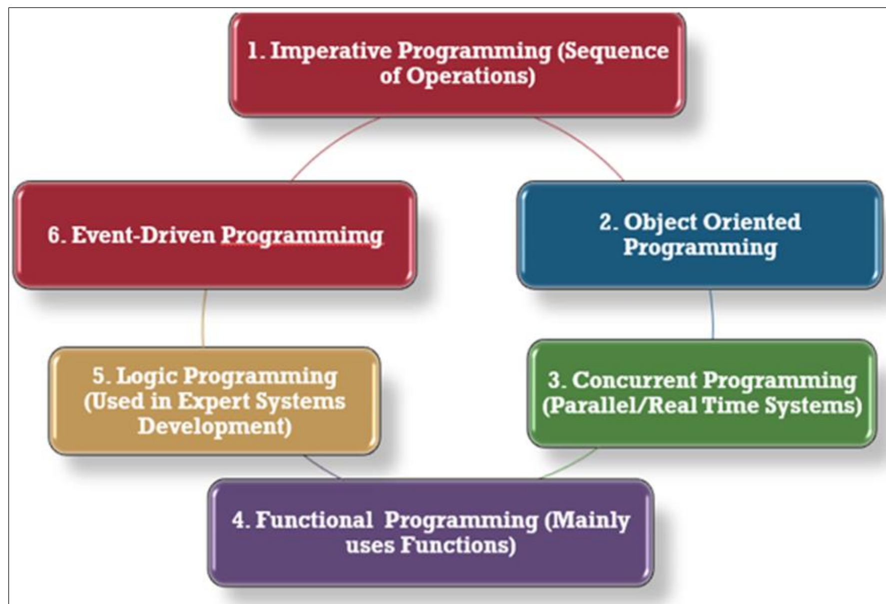


Figure 1 Major Programming Paradigms

Object Oriented Programming in Python focuses on creating reusable code, a concept commonly known as an acronym DRY (Don't Repeat Yourself) [12]. Again, the source code resulting from object-oriented programming is highly maintainable due to the modular approach applied. Moreover, program correctness could be enhanced in OOP since every class has a specific task. Thus, an error in one part of the code, could be rectified locally without having to affect other parts of the code. Some of the concepts to be studied and practically demonstrated in this work are Class, Object, Inheritance, Constructors, among others.

2. Class and object implementation

It is important to begin this section with a definition of the concept of Class [13]. In OOP, a class is a user defined blueprint or prototype [14] from which objects are created. In other words, a class represents the set of properties or methods that are common to all objects of one type. In practice, one of the first tasks at this point is to understand how to create a class. A Python Class is created using the following syntax:

class ClassName:

#Statement Suite

As shown in the syntax statement, class creation involves using the keyword [15] class followed by the name of class, followed by a colon, then a suite of statements. There are important rules to be observed, to avoid syntax errors. These rules are enumerated in Table 1.

Table 1 Class Creation Rules

RULE 1	The keyword “class” should be in lower case, while the class name should start with a capital letter [16].
RULE 2	The statement suite (body of the class) consists of a number of statement types, ranging from fields (properties), constructors, functions, pass statements, among others, as will be further explained.
RULE 3	The body of the class starts on a new line, indented one tab from the left.
RULE 4	After creating a class, it has to be instantiated into objects.

Before a programmer can use a class, it has to be instantiated [17]. It is through this process that an object gets created from a class. The syntax to create the instance of the class is as follows:

ObjName = ClassName (arg)

Where ObjName = name of the object being created.

ClassName = name of the class being created

arg = arguments.

2.1. Practical Illustration One

Fig. 2 depicts a practical illustration of the creation of a Python object called emp from the class Employee, where emp object inherited the fields (attributes) of the Employee class, and an assignment statement created using emp.name.

```
class Employee:
   id = 10
   name = "John"

emp = Employee()
emp.name="Samuel"
```

Figure 2 Code for Illustration of Class and Object Creations

3. Inheritance implementations

The concept of Inheritance in OOP allows a programmer to define a class that takes and manifests the methods and properties from another class. The parent class is the class being inherited from, and this is also known as the base class [18]. On the other hands, a child class is the class that inherits from the parent or base class. Another name for the child class is the derived class [19].

There are three major types of inheritances supported by Python. These are Single Level Inheritance [20], Multi-Level Inheritance [21] and Multiple Level Inheritance [22], all of which will be illustrated in this work.

The general syntax for implementing inheritance is as follows:

Child Class (Parent Class)

Where Child Class = name of the child class.

Parent Class = name of the parent class.

3.1. Practical Illustration Two

Given that a Python program is made up of two classes called Sun and Moon. Create the two classes, then write a line of statement to show that Moon inherits from the Sun. The constructor method should be populated with a pass statement [23]. The source code for solution to this problem is shown in Fig. 3.

```
#SOLUTION:

# sun_moon.py

class Sun:

    def __init__(self):

        pass

class Moon (Sun):  # Inheritance statement

    def __init__(self):
```

Figure 3 Code for Practical Illustration Two

3.2. Practical Illustration Three

A multiple inheritance design has eight classes A, B, C, D, E, F, G and H where B, C, D inherited A; E inherited B and C; F inherited B and D; G inherited C and D while H inherited E, F and G. The task to be accomplished in this regard is to draw the inheritance diagram, and to write a simple program to accomplish this. The first solution for this problem is shown in Fig. 4.

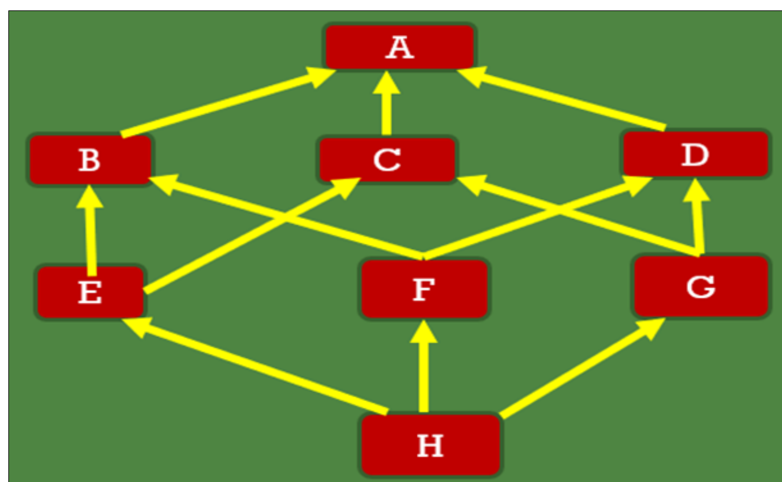


Figure 4 The Inheritance Diagram

The python code equivalence of the above diagrammatic representation [24] is shown in Fig. 5. As shown in the code segment, the class on the topmost hierarchy is class A, which was inherited by another class B. This is indicated by placing the parent A within a bracket under B. This thread continues, until the last inheritance statement, where H inherits from the parent classes E, F and G. The three major inheritance types are clearly annotated in the right-hand side (RHS) of the diagram. A single inheritance is a simplistic case, where one class inherits another, for instance B (A). In a multi-level inheritance, there is a trail of inheritance, with one giving rise to another, for instance C(A) followed by E(C), and so on. Finally, in a multiple inheritance case, a particular class inherits from multiple parents at the same time, for instance, H (E, F, G).

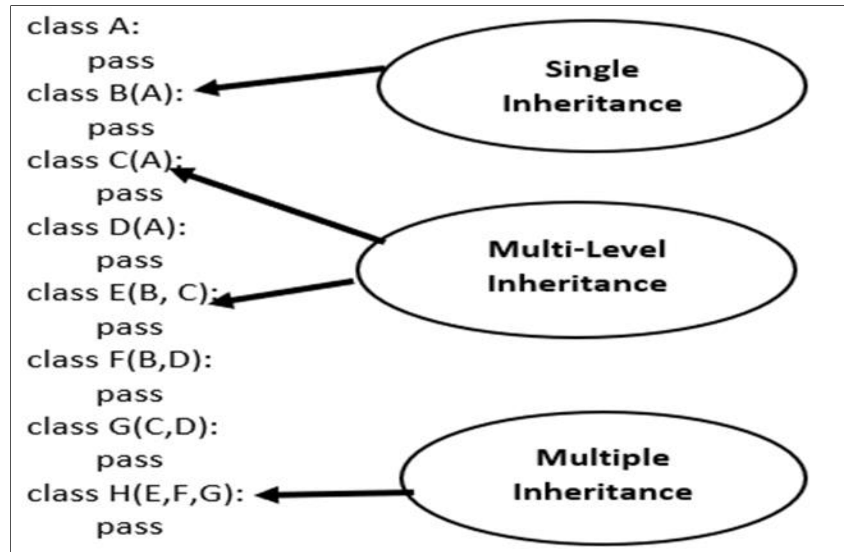


Figure 5 The Inheritance Source Code

4. The self-argument implementations

Fig. 6 is a code segment [25] having a given class and a single method. The class name is Example, which is followed by an indented block [26] of statements which represent the body of the class. The name of the method defined is test.

```
# Illustrating a Simple Class.
class  Example:
        # A Simple Method
        def  test (self):
                print ("Hello INSY 404")
# Program Calls
obj=Example( )
obj.test()
```

Figure 6 The Inheritance Source Code

REAL LIFE FUNCTIONS:	PYTHON METHODS
Function Definition:	**Method Definition:**
MathF(Para1, Para2,...ParaN)	PythM(Self, Para2,...ParaN)
Function Call:	**Method Call:**
X=MathF(Arg1, Arg2,...ArgN)	X=PythM (Arg2,...ArgN)

Figure 7 Disparity in Mathematical Functions and OOP Method Calls

There are a number of key observations in the source code illustration in Fig. 6. First is that it is usual for class methods to have an extra first parameter known as "self" in the method definition. In real life, the number of parameters [27] in a typical mathematical function definition is expected to tally with number of arguments in function calls [28]. But this rule is apparently defied in OOP implementations in Python. This syntactical variation is shown in Fig. 7.

Thus, the first parameter is not given value (arguments) during method calls, but rather is usually provided internally by Python. There are a number of important points to bear in mind when implementing the self-arguments, some of which are itemized in Table 2:

Table 2 Key Points on Self Argument

Key Points	Narration
Point 1	Even if a method takes no argument during its call, a programmer still has to insert one parameter - the self – as shown in the last illustration test( ).
Point 2	The Point 1 above is similar to "this" pointer in C++ and "this" reference in Java.
Point 3	When the programmer calls a method of the object as *myobj.method (arg1, arg2)*, this is automatically converted by Python into *MyClass.method (myobj, arg1, arg2).*
Point 4	In summary, it follows that self is a special parameter in Python [29].

5. Constructor implementations

A constructor is defined as a special method used for initializing the instance variables during object creation. A Python Constructor is usually implemented using __init__ (self) [30]. There are three major types constructors as shown in Fig. 8, though a number of researchers limit their interest to only two.

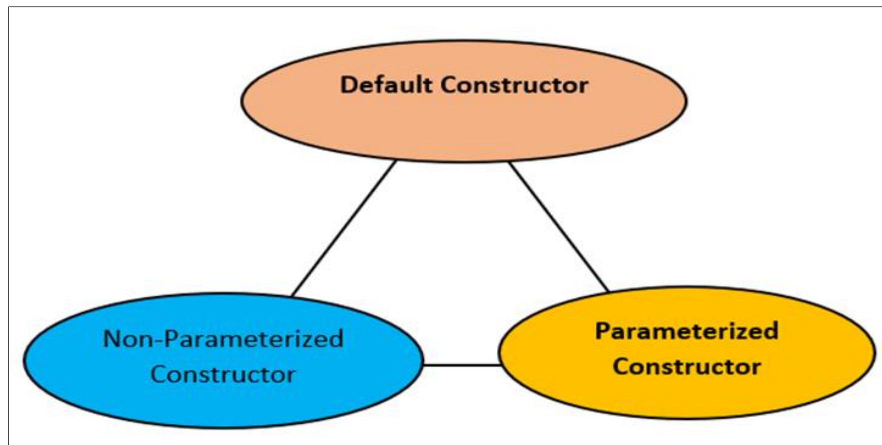


Figure 8 Three Types of Constructors

For a very unambiguous explanation of the implementation [31] of the three types of constructors, it is necessary to use some practical illustrations. The flow diagram shown in Fig. 9 shows how to implement a default constructor [32]

The default constructor specifies default values in the Constructor definition, and at the same time, provides parameter equations within the __init__ method block. The flowchart [33] can be explained using the following IF statements:

IF no arguments are specified while creating an instance of the class, THEN the default values set for the __init__() method parameters gets assigned to instance, OTHERWISE IF you specify arguments during instantiation, THEN the default values assigned to __init__() method parameters will be overwritten with the latest values. Having looked at the default constructor, it is important to mention that parameterized constructor [34] implementations come with explicit parameters, while the in non-parameterized [35] case, there are no explicit parameters, rather, the self-parameter is prominently featured. This research will give practical illustrations at this moment.

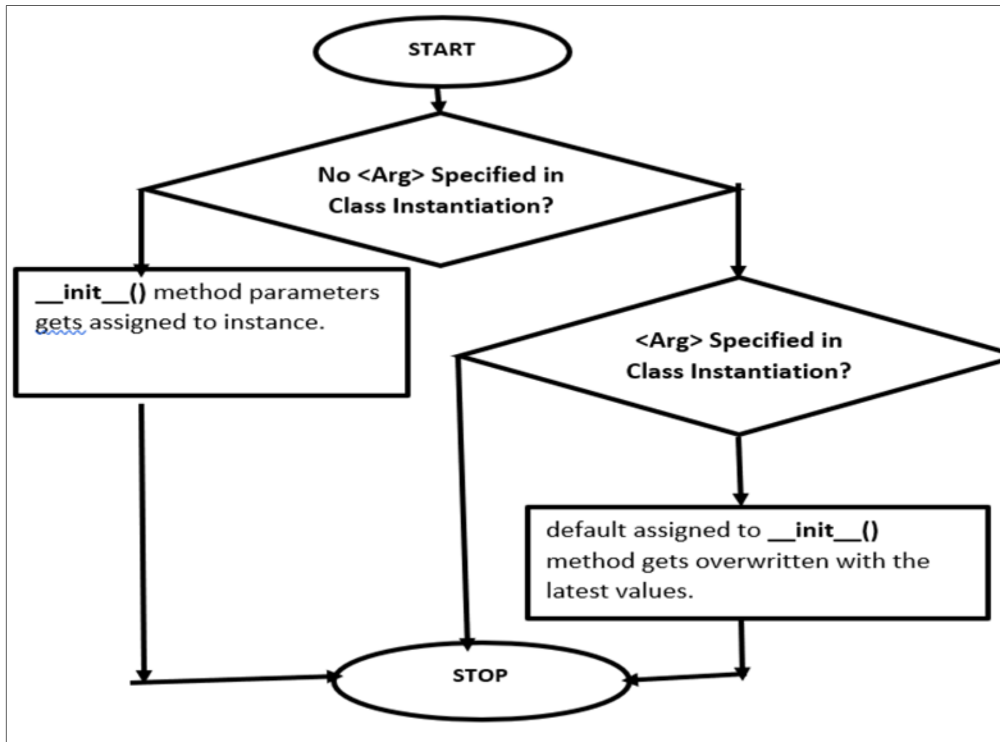


Figure 9 Flowchart for Default Constructor

5.1. Practical Illustration Four

The aim of the practical illustration is to create a parameterized constructor for *farm fruits*. The parameter is called *fruitname*. The Fruit class has two other methods. The first one queries a person for the type of fruit he/she loves, and another responds with "*I Love xyz*" where xyz is the actual fruit name supplied as an argument to method invocation. The class should be instantiated accordingly. The solution is shown in Fig. 10.

SOLUTION:

```
class Fruit:    #fruit_para.py
  def __init__(self, fruitname):
    print("Demonstrating Parameterized Constructor")
    self.fruitname=fruitname
  def queryfruit(self):
    print("Which Fruit do you Love?")
  def sayfruit(self):
    print("I Love " + self.fruitname)
objfru=Fruit("BANANA")
objfru.queryfruit()
objfru.sayfruit()
print(objfru.fruitname)
```

PARAMETERS

1. Parameter Definition
2. Parameter Equation
3. Instantiation MUST be Argumented.
4. Invocation

Figure 10 Parameterized Constructor Solution

As shown, the solution has been annotated [36], with four arrows originating from the right-hand side (RHS) and pointing to the lines of the codes that represent definite solutions. Also, as shown in the comment section [37] of the first line of the code, the name of the python file is *fruit_para.py*.

5.2. Practical Illustration Five

The aim of the practical illustration is to create a non-parameterized constructor for farm fruits. As in parameterized case, the Fruit class has two other methods. The first one queries a person for the type of fruit he/she loves, and another responds with "*I Love xyz*" where xyz is the actual fruit name, defined within the constructor as an instance variable. The class should be instantiated accordingly. The solution is shown in Fig. 11.

SOLUTION:

```
class Fruit:   #fruit_nonpara.py
  def __init__(self):
    print("Demonstrating Non-Parameterized Constructor")
    self.fruitname="ORANGE"
  def queryfruit(self):
    print("Which Fruitdo you Love?")
  def sayfruit(self):
    print("I Love  " + self.fruitname)
objfru=Fruit()
objfru.sayfruit()
print(objfru.fruitname)
```

NON-PARAMETERIZED
1. No explicit Parameter defined.
2. Instance Variables explicitly defined
3. Instantiation
4. Invocation
5. In this case, the fruit is ORANGE.

Figure 11 Non-Parameterized Constructor Solution

As shown, the solution has been annotated, with five arrows originating from the right-hand side (RHS) and pointing to the lines of the codes that represent definite solutions. Also, as shown in the comment section of the first line of the code, the name of the python file is *fruit_nonpara.py.*

5.3. Practical Illustration Six

The aim of the practical illustration is to combine both parameterized and non-parameterized constructors to form default constructors to create a new Fruit class that makes use of Default Constructor. The default fruit name should be *APPLE*. The solution is shown in Fig. 12.

SOLUTION:

```
class Fruit:   #fruit_default.py
  def __init__(self, fruitname="APPLE"):
    print("Demonstrating Default Constructor")
    self.fruitname=fruitname
  def queryfruit(self):
    print("Which Fruit do you Love?")
  def sayfruit(self):
    print("I Love " + self.fruitname)
objfru=Fruit("BANANA") #Use of Parameterized Call
objfru.queryfruit()
objfru.sayfruit()

#Second Case - Use of Default Values
objfru=Fruit()
objfru.queryfruit()
objfru.sayfruit()
```

Figure 12 Default Constructor Solution

6. The concept of overriding

One of the important concepts in Object Oriented Programming is the overriding. This will be explained, and practically implemented as follows. A child class naturally inherits the methods and attributes of the parent class. However, if the child class has a method, which has same name as that of a parent class, but with a different definition/content, then the child's method is said to override [38] that of the parent. This is comparable to stubborn children who refuse to look like their parents/ancestors, but chose to be unique, strange and weird !

6.1. Practical Illustration Seven

The aim of this section is to create two Python Programs called *b4overide.py* and *afteroveride.py* respectively, so as to illustrate Overriding in Python. These are shown in Fig. 13 and Fig. 14. In the first code segment, there are two classes designated as *King* and *Prince*, where the child is naturally open to inherit attributes of the parent. However, in the second segment, the child overrides the parents. This is clearly activated using the action method, which gives two major outputs. These are "*There is Trouble in the Land*" and "*The Small Boy Prince Overrides Papa King*" respectively.

SOLUTION:

```
# b4overide.py
class King:
  def action (self):
    print ('The Command of the King Stands')
class Prince (King):
  pass
x=Prince ()
x.action()
```

NOTE: The Prince class is has no active method yet (see the pass statement), so no override.

Figure 13 Classes Before Override

SOLUTION:

```
# afteroveride.py
class King:
  def action (self):
    print ('The Command of the King Stands')
class Prince (King):
  def action (self):
    print ('There is trouble in the Land')
    print('The Small Boy Prince now Overide Papa King')
x=Prince ()
x.action()
```

NOTE: Here, a definition of a method called **action** overrides the first **action** method in King class.

Figure 14 Classes After Override

7. The diamond problem

Before concluding this work, it is important to explore what is usually known as the Diamond Problem, explain what it is all about, and state how it is solved. In doing this, four distinct classes A, B, C and D will be utilized, where A is the topmost in the hierarchy, and the contentious method is called "methodxyz".

A Diamond Problem [39] is an issue resulting from multiple inheritance of four classes in OOP as explained in the following scenario. If classes B and C inherit from A and class D inherits from B and C, and both B and C have a method called methodxyz. The issue then is, "Which of the competing methodxyz will D inherit?" This ambiguity is known as the diamond problem. A sample code as well as pictorial diagram to illustrate the Diamond Problem are shown in Fig. 15.

Different OOP languages resolve the diamond problem puzzle in different ways. In Python it is resolved using what is known as MRO (Method Resolution Ordering) algorithms [40]

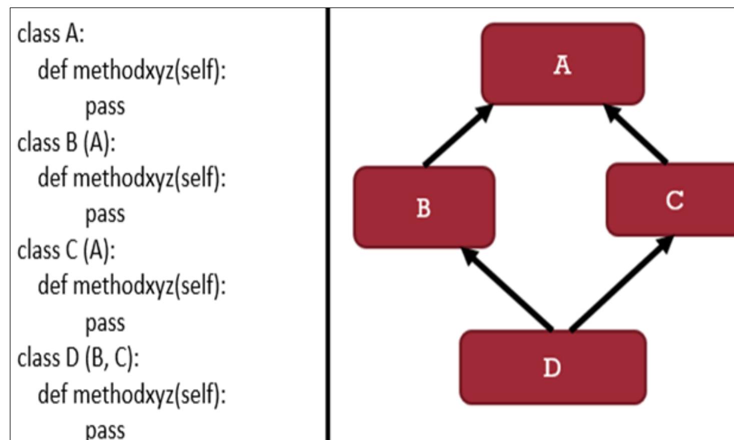


Figure 15 Diamond Problem – Code and Diagram

8. Conclusion

This work has presented a conceptual and practical oriented study of Object-Oriented Programming (OOP). Effort was made to present a number of fundamental concepts, referred to in this work as the backbone concepts. The concepts covered are class and object creation, inheritance, constructors, mathematical and OOP method disparity, and the self-concepts. Other backbone concepts covered are overriding between the parent and child class, the Damond Problem and resolution. In order to enhance understanding of this subject matter, about ten code segments were shown. There were also pictorial presentations, flow diagrams, among others. It is hoped that this work will be very useful to researchers and practitioners in Object Oriented implementations.

Compliance with ethical standards

Acknowledgments

The authors acknowledge themselves for a successful collaboration.

Disclosure of conflict of interest

The Authors of this paper, Monday Eze, Charles Okunbor and Collins Umoke certify that they have no conflict of interest.

References

[1] Asha R, Kavana M, Parvathy S and Shreelakshmi C. Object Oriented Programming and its Concepts. International Journal for Scientific Research and Development. 2017; 5(9): 840-842.

[2] Dmitrii R. The C++ programming language in cheminformatics and computational chemistry , Journal of Cheminformatics. 2020; 12(10): 1-16.

[3] Veysel S. Association of cognitive, affective, psychomotor and intuitive domains in education, Sonmez Model. Universal Journal of Education Research. 2017; 5(3): 347-356.

[4] Rafael B, Lars B, Mehdi D, Amal E, Jorge G, Joao L, Gregory O, Alexander P and Alessandro R. A Survey of Programming Languages and Platforms for Multi-Agent Systems, Informatica. 2006; 30(1): 33–44.

[5] Peter W, Karina G, Shirish G, Ashwin R, Ingmar HR. Interactive Programming Paradigm for Real-time Experimentation with Remote Living Matter. In Proceedings of the National Academy of Sciences of the United States of America (PNAS). 19 March 2019; 116 (12): 5411-5419.

[6] Pengyu N, Marinela P, Zhiqiang Z, Sarfraz K, Aleksandar M, Milos G. Unifying Execution of Imperative Generators and Declarative Specifications. Proc. ACM Program. Lang. 2020; 4(217): 1-26.

[7] Isaiah A. Technological Advancement in Object Oriented Programming. International Journal of Applied Engineering Research. 2019; 14(8): 1835-1841.

[8] Keshta IM. Software Refactoring Approaches: A Survey. International Journal of Advanced Computer Science and Applications. 2017; 8(11): 542–547.

[9] Brown C, Loidl HW, Hammond K. Paraforming: Forming Parallel Haskell Programs Using Novel Refactoring Techniques. In Proceedings of International Conference on Trends in Functional Programming, Madrid, Spain, 16 May 2011; 82–97.

[10] Bahaa M. Petri Net Based Event Driven Programming. Journal of Engineering and Applied Sciences. 2019; 14(13): 4295-4302.

[11] Nimit T, Abhilash S. Python as Multi Paradigm Programming Language. International Journal of Computer Applications. 2020; 177(31): 38-42.

[12] Onofrei G, Ferry P. Reusable Learning Objects: A Blended Learning Tool in Teaching Computer-aided Design to Engineering Undergraduates, International Journal of Educational Management. 2020; 34(10): 1559-1575.

[13] Kirby M, Samuel S, Brian R, Stuart W. Java vs Python Coverage of Introductory Programming Concepts: A Textbook Analysis. Information Systems Education Journal. 2017; 15(3): 4-13.

[14] Alam M. Prototyping Mass Nouns from Students' Perspective: A Pedagogical Implication. Advances in Social Sciences Research Journal. 2020; 7(1): 76-99.

[15] Xing C, Hans L, Halvard M. On the performance of the Python programming language for serial and parallel scientific computations. Scientific programming. 2005; 13(1): 31-56.

[16] Sebastian B. A Primer on Python for Life Science Researchers. PloS Computational Biology. 2007; 3(11): 1-6.

[17] Bogdanchikov A, Zhaparov M, Suliyev R. Journal of Physics: Cconference Series. 2013; 423(1): 1-6.

[18] Ashwin U. Object Oriented Programming and its concepts. International Journal of Innovation and Scientific Research. 2016; 26(1): 1-6.

[19] Berk E, Charles M, Cameron M. PloS Computational Biology. 2016; 12(6): 1-43.

[20] Fawzi A, Amjad M. A comparative study on the effect of multiple inheritance mechanism in java, c++, and python on complexity and re-usability of code. International Journal of Advanced Computer Science and Applications. 2017; 8(6): 109-116.

[21] Mehul P, Modi R, Pillai S. Inheritance and its types in Object Oriented Programming Using C++. International Journal of Recent Technology and Engineering. 2019; 8(4): 1991-1998.

[22] Vishal M, Yash N. Review on concepts related to Object-Oriented programming system. Iconic research and engineering journals. 2019; 3(6): 56-58.

[23] Preetee K, Gaurishamkar G. Programming language python: A review. International Journal of Advanced Research and Innovative Ideas in Education. 2020; 6 (2): 1634-1637.

[24] Yang L. Content analysis of the diagrammatic representation of primary science textbooks. Eurasia Journal of Mathematics, Science and Technology Education. 2016; 12(8): 1937-1951.

[25] Jitesh D. Understanding Code Patterns – Analysis, Interpretation and Measurement. International Journal of Computer and Electrical Engineering. 2009; 1(1): 46-55.

[26] Radoslava K, Velin K, Dafina K. A Methodology for the Analysis of Block-based Programming Languages appropriate for children. Journal of Computer Science and Engineering. 2019: 13(1): 1-10.

[27] Lei Y. Evaluation of Three Methods for Estimating the Weibull Distribution Parameters of Chinese Pine. Journal of Forest Science. 2008; 54(12): 566-571.

[28] Zeraoulia R, Alvaro S, David O. A New Special Function and Its Application in Probability. International Journal of Mathematics and Mathematical Sciences. 2018; 18(5146794): 1-12.

[29] Daniel F. Use of python Programming Language in Astronomy and Science. Journal of Computational Interdisciplinary Sciences. 2018; 3(3): 1-11.

[30] Uolevi N, Jorma S, Matti T, Stuart W. Python and Roles of Variables in Introductory Programming: Experiences from Three Educational Institutions. Journal of Information Technology Education – Research. 2007; 6(1): 199-214.

[31] Wahyudin D, Ali M, Rinda C, Abdusy A. Strategic Design of Information System Implementation at University. International Journal of Engineering and Technology. 2018; 7(2): 787-791.

[32] Lone L, Bent T, Kurt N. Computational Abstraction Steps. Journal of Object Technology. 2010; 9(6): 1-23.

[33] Carlos R. Flowchart for predictive use of the Impella. Interventional Cardiology Review. 2017; 12(2): 13-14.

[34] Shilagram P. Parameterized Key Diffusion Approach of AVK Based Cryptosystem. Covenant Journal of Informatics and Communication Technology. 2017; 5(1): 1-19.

[35] Probal C, Anil G, Hannu O. Classification Based on Hybridization of Parametric and Non-parametric Classifiers. IEEE Transactions on Pattern Analysis and Machine Intelligence. 2009; 37(7): 1153-1164.

[36] Abbott A, Cyranoski D, Jones N, Maher B, Schiermeier Q, Van R. Metrics: do metrics matter?, Nature. 2010; 465(7300): 860-862.

[37] Arfon S, Kyle N, Daniel K, Lorena B. Journal of Open Software: Design and first year review. PeerJ Computer Science. 2017; 4(1): 1-19.

[38] Antoine B. Method overloading and overriding cause encapsulation flaw. 21st ACM Symposium on Applied Computing. Held in Dijon France. April 2006; 1424-1428.

[39] Rajesh J, Anil B, Satyakam P. Diamond Effect in Object Oriented Programming Languages. International Journal of Computer Science and Information Technologies. 2015; 6(3): 3104-3111.

[40] Meszarosova E. Is Python an Appropriate Programming Language for Teaching Programming in Secondary Schools? ICTE Journal. 2015; 4(2): 5-14.