**definition**

Object-oriented programming (OOP) is a way of designing software that focuses on "objects" instead of just functions or logic. Each object combines data (attributes) and operations (methods) that can be performed on that data.

**history**

OOP originated in the 1960s with the Simula programming language, developed by Ole-Johan Dahl and Kristen Nygaard, which introduced the concepts of classes and objects. It gained widespread use in the 1980s, particularly with the Smalltalk language that fully implemented OOP principles.

**Evolution**

1970s: The development of Smalltalk by Alan Kay, Dan Ingalls, and Adele Goldberg at Xerox PARC further advanced OOP concepts. Smalltalk was one of the first languages to fully embrace OOP, introducing features like dynamic typing and a rich graphical user interface.

1980s: The popularity of OOP grew with the introduction of languages like C++ by Bjarne Stroustrup, which combined the efficiency of C with OOP features. This decade also saw the emergence of Objective-C, which added OOP capabilities to the C language.

1990s: Java, developed by James Gosling and his team at Sun Microsystems, became a significant player in the OOP landscape. Java's "write once, run anywhere" philosophy and its robust standard library made it a popular choice for enterprise applications.

2000s: OOP continued to evolve with languages like C#, developed by Microsoft, and Python, which embraced OOP principles while maintaining simplicity and readability. Modern languages like Ruby and Swift also incorporate OOP features, further expanding the paradigm's reach.

**Key Contributors to OOP Development**

Ole-Johan Dahl and Kristen Nygaard: Creators of Simula, the first object-oriented programming language.

Alan Kay: Developed Smalltalk and coined the term "object-oriented programming."

Bjarne Stroustrup: Creator of C++, which popularized OOP in systems programming.

James Gosling: Lead developer of Java, which brought OOP to a broader audience.

**Notable languages that support OOP**

Many programming languages, like C++, Java, Python, and Ruby, have since adopted and evolved OOP concepts. Notable figures in OOP's history include Alan Kay, who popularized the term "object-oriented programming," and Bjarne Stroustrup, who developed C++ to add OOP features to the C programming language.

**Comparison of Object-Oriented Programming (OOP) with Procedural Programming**

**Introduction**

Programming paradigms form the foundation of software development methodologies, providing developers with different ways to organize and manage code. Two significant paradigms are Object-Oriented Programming (OOP) and Procedural Programming. This report compares and contrasts these paradigms, discussing their advantages and disadvantages and providing insights on appropriate scenarios for each approach. Additionally, I’ll provide an overview of popular OOP languages, with a special focus on Python, highlighting its key features that support OOP.

**Topic 2: Comparison with Procedural Programming**

**Definition**

* **Procedural Programming**: This paradigm follows a linear top-down approach, where programs are structured as a series of procedures or routines. The primary focus is on functions and the sequencing of task execution.
* **Object-Oriented Programming**: In contrast, OOP is centred around objects, which are instances of classes containing data (attributes) and methods (functions). The focus is on data encapsulation, modularity, and the relationships between objects.

**Advantages and Disadvantages**

| **Aspect** | **OOP Advantages** | **OOP Disadvantages** | **Procedural Advantages** | **Procedural Disadvantages** |
| --- | --- | --- | --- | --- |
| **Modularity** | Encourages modular design, making code reusable. | Can become complex for small programs. | Simple and straightforward for small scripts. | Less modular, making code harder to reuse. |
| **Encapsulation** | Data hiding increases security and reduces dependency. | Overhead in memory and performance. | More efficient in terms of memory usage. | Maintenance can become challenging as programs grow. |
| **Inheritance** | Facilitates code reuse and extension. | Can lead to inappropriate inheritance. | Easier to understand for novices with simple tasks. | Lacks abstraction: global state can lead to unexpected behaviour. |
| **Polymorphism** | Allows flexibility in function usage. | Can lead to excessive complexity if misused. | Straightforward function calls with predictable results. | Can lead to code that is less adaptable to change. |

**Suitable Scenarios for Each Paradigm**

* **When to Prefer OOP**:
  + **Large-Scale Applications**: OOP is more favoured for large systems due to its modularity and maintainability. For instance, enterprise applications like Customer Relationship Management (CRM) systems often use OOP languages to encapsulate business logic.
  + **Software Development Requiring Frequent Updates**: Systems that evolve based on user feedback benefit from OOP’s modular structure, allowing adjustments without overhauling the entire codebase.
* **When to Prefer Procedural Programming**:
  + **Simple Scripting Tasks**: Procedural programming is suitable for simple scripts and smaller applications where overhead from OOP is unnecessary.
  + **Performance-Critical Applications**: Programs needing high performance, such as gaming engines or embedded systems, can benefit from the efficiency of procedural code.

**Topic 3: Overview of OOP Languages**

**Overview of Popular OOP Languages with Focus on Python**

Among the various object-oriented languages, Python has gained immense popularity due to its simplicity and versatility.

**Key Features of Python that Support OOP**

1. **Class and Object Creation**: Python allows users to create classes and instantiate objects easily. The syntax is straightforward, reducing the learning stress for newcomers.
2. class Dog:
3. def \_\_init\_\_(self, name):
4. self.name = name
5. def bark(self):
6. return f"{self.name} says woof!"
7. my\_dog = Dog("Buddy")
8. print(my\_dog.bark()) # Output: Buddy says woof!
9. **Inheritance**: Python supports class inheritance, allowing the creation of new classes based on existing ones, promoting code reuse.
10. class Animal:
11. def speak(self):
12. return "Animal sound"
13. class Cat(Animal):
14. def speak(self):
15. return "Meow"
16. my\_cat = Cat()
17. print(my\_cat.speak()) # Output: Meow
18. **Polymorphism**: Python supports polymorphism, enabling a single interface to support different underlying data types. This is achieved through method overriding, where derived classes can redefine methods from base classes.
19. **Encapsulation**: Python provides mechanisms to enforce encapsulation, restricting access to certain parts of an object to protect the internals from unintended interference (e.g., through the use of private attributes).
20. **Dynamic Typing**: Python’s dynamic typing enhances flexibility, allowing developers to write adaptable and scalable code. This feature allows for quicker prototyping and development.

**Conclusion**

In summary, while both Object-Oriented Programming and Procedural Programming have their unique strengths and weaknesses, the choice between them often depends on the specific needs of a project. OOP shines in complex and large-scale applications due to its focus on modularity and reuse, while procedural programming remains effective for straightforward tasks and performance-driven applications. Python exemplifies a modern OOP language that combines simplicity and robust object-oriented capabilities, making it a popular choice among developers.

**References**

1. Meyer, B. (1997). *Object-Oriented Software Construction*. Prentice Hall.
2. Lutz, M. (2021). *Learning Python*. O'Reilly Media.
3. Stroustrup, B. (2013). *The C++ Programming Language*. Addison-Wesley.
4. Sande, S. (2019). *Python for Data Analysis*. O'Reilly Media.

**Programming task**

class Car:

def \_\_init\_\_(self, make, model, year, color):

self.make = make

self.model = model

self.year = year

self.color = color

def display\_details(self):

"""Display the details of the car."""

print(f"Car Details:\nMake: {self.make}\nModel: {self.model}\nYear: {self.year}\nColor: {self.color}")

def update\_color(self, new\_color):

"""Update the color of the car."""

self.color = new\_color

print(f"The color of the car has been updated to: {self.color}")

# Task 2: Instantiate multiple objects of the Car class

car1 = Car("Toyota", "Corolla", 2020, "Blue")

car2 = Car("Honda", "Civic", 2019, "Red")

car3 = Car("Ford", "Mustang", 2021, "Black")

# Display the details of each car

car1.display\_details()

car2.display\_details()

car3.display\_details()

# Update the color of one car object and display updated details

car1.update\_color("Green")

car1.display\_details()

# Task 3: Create a simple program to input details for a new car

def create\_car\_from\_input():

make = input("Enter car make: ")

model = input("Enter car model: ")

year = input("Enter car year: ")

color = input("Enter car color: ")

new\_car = Car(make, model, year, color)

print("\nNew Car Created:")

new\_car.display\_details()

# Initiating the car creation process from user input

create\_car\_from\_input()