

Object Oriented Programming

”
“ *“The assumption that the problem of love is the problem of an object , not the problem of a faculty . People think that to love is simple, but that to find the right object to love - or to be loved by- is difficult.” - Erich Fromm*

1.1. What is a Program?



A set of instructions or a sequence of commands given to a computer to accomplish a specific task is called **Program**. Tasks can be anything from a simple calculation to complex operations. It may involve interaction with end users or other software systems.

Programming languages help us to write instructions to the computer in a human readable language. Instructions once written, are compiled, and translated into machine code, which consists of low-level instructions, that are understood and performed directly by computer's hardware.

1.2. Pillars of Programming

Enterprise Software programs are centered on two elements: code and data, based upon which the organization and construction of a program can be approached: **process-oriented** (code) and **object-oriented programming** (data).

The **process-oriented** model focuses on the sequence of steps or instructions (code) that a program follows. In this model, the program is seen as a series of linear steps, with code acting on data. A program is structured as a collection of independent processes that communicate and coordinate with each other to achieve the desired functionality. Each process is responsible for performing a specific task, and they interact by passing messages or sharing resources.

Imagine we are developing a simple video game where the characters move, shoot, and collect coins. In a process-oriented approach, program will be divided into separate processes, each responsible for one aspect of the game:

Movement Process: This process handles the character's movement when the player presses the arrow keys.

Shooting Process: This process deals with shooting bullets when the player presses the spacebar.

Coin Collection Process: This process manages the character's interaction with coins and keeps track of the collected coins.

These processes run independently and communicate with each other as needed. For instance, when the character collects a coin, the Coin Collection Process informs the game's score display process to update the player's score. When the character shoots a bullet, it informs the Shooting Process to create a new bullet object on the screen.

This approach helps to organize and manage the game's different functionalities separately, making it easier to understand and modify each part of the game without affecting the others. If we want to improve the shooting mechanism, we can focus on the Shooting Process without altering the Movement or Coin Collection processes.

Procedural languages like C are commonly used in this approach. It is used in systems where concurrency and asynchronous operations are crucial, especially while designing Operating Systems or

networking systems. However, as programs grow larger and more complex, problems may arise due to the increasing difficulty of managing and understanding the codebase.

The applicability of Process oriented programming is limited, especially to systems heavily dependent on concurrency or parallelism. They are not best suited for enterprise application development, as the goal is to best represent the enterprise domain in a virtual world and process-oriented approach does not provide expressiveness in data modeling. Procedural languages don't capture active aspects of the domain, even though they do support complex data types. Programs written in procedural languages like C, Fortran, etc. are just a series of technical manipulation of data. They fail to capture any higher level meaning specific to the domain.

Enterprise applications involve wide range of functionalities and use of independent processes can make the life of a developer hell, as managing inter process communication can turn out to be a very challenging task. Memory usage is another consideration, as process-oriented approach tends to consume more resources due to the need for separate memory spaces and process management. Enterprise applications often need to efficiently manage their resources to cater a large number of users or transactions.


To address the challenges of increasing complexity, **object-oriented programming (OOP)** was developed. OOP organizes a program around its data, represented by objects which are real world entities, and the interactions between those objects (real world entities). In this paradigm, an object is an instance of a class, which defines the properties (data) and behaviors or functions (code) that objects of that class possess. The program's structure is defined by the relationships and interactions between these objects (real world entities), which exist in memory.

Imagine we need to develop a comprehensive system to manage various types of vehicles, including cars, bikes, and trucks, as well as their individual components, such as engines and brakes. Each vehicle type has unique attributes and behaviors, and each component plays a specific role in the operation of the vehicle.

Class Definitions:

- 1. Vehicle Class: Our base class for all vehicles, encompassing common attributes like brand, model, and color. It includes methods for starting and stopping the vehicle.*
- 2. Car, Bike, and Truck Classes: These are subclasses of the Vehicle class, each customized to suit the unique characteristics of its respective vehicle type. They include methods specific to their behaviors.*
- 3. Vehicle Part Classes: We can define separate classes for key vehicle components (attributes of vehicle sub classes), such as engines and brakes. These classes encapsulate the attributes and behaviors of individual vehicle parts.*

As Eric Evans mentions in his book, Domain Driven Design, “Object-oriented programming is powerful because it is based on a modeling paradigm, and it provides implementations of the model constructs”.

 **OOP** provides several advantages over the process-oriented model. It promotes modularity, encapsulation, and reusability by bundling data and code together in self-contained objects. Objects have well-defined **interfaces**, specifying how other parts of the program can interact with them, while keeping their internal details hidden. This **abstraction** allows for better code organization, easier maintenance, and the ability to model complex real-world systems more accurately. It is a natural fit for expressing domain level concepts across the programs and is often the first choice of architects while implementing model driven design across the application development.

In Object oriented languages, the paradigm (a fundamental approach or way of thinking about and solving problems) is logic, and the model, a representation or abstraction of a system, concept, or domain, is created by defining a set of logical rules and facts that describe the relationships, constraints, and behavior of the entities within the system or domain being modeled.

While process-oriented approach can be scalable by adding more processes, in contrast OOP compensates scalability through the use of design patterns and architecture principles like microservices or event driven design. OOP's modular nature allows different components of an enterprise application to be developed, deployed, and scaled independently.

By organizing a program around its data and defining clear interfaces to access and manipulate that data, object-oriented programming provides a more structured and manageable approach to software development. It offers benefits such as code reusability, extensibility, and easier maintenance, making it particularly useful for larger and more complex projects. These tools allow developers to model real-world entities more naturally, making it easier to represent and manipulate complex data structures. ***So, what are Objects in Object Oriented Programming?***

1.3. Objects

In an enterprise world of software programs, an **Object is a collection of data and associated behaviors**. When dealing with real world problems, we can encounter many objects while designing a system. Let us take an example:

Suppose we are developing a social media application. In this application, we have two main types of objects: User and Post.

User: The User object represents an individual user of the social media platform. It would have associated data such as the user's name, email, date of birth, and profile picture. Additionally, it would have behaviors such as logging in, updating profile information, and posting content.

Post: The Post object represents a post or message shared by a user on the social media platform. It would have associated data such as the content of the post, the user who created it, the timestamp, and the number of likes or comments. The behaviors associated with a post could include editing the content, deleting the post, or interacting with other users' posts through comments or likes.

In this example, we differentiate between the User and Post objects based on their distinct characteristics and functionalities. The User object focuses on representing and managing user-related data and behaviors, while the Post object deals with the content and interactions between users' posts. By modeling these objects separately, we can maintain a clear separation of concerns and establish different sets of properties and methods for each object type. This separation allows us to perform operations specific to users or posts without mixing their functionalities.

1.4. Classes

To define what kind of object we are dealing with, we use Classes. In a car showroom, we have different models of car, but they all have attributes such as wheels, steering, headlights, engine, etc. and behaviors such as driving, honking, etc. associated with one class: CAR, a general class of cars.



*A **Class** serves as a blueprint or template that defines the structure, attributes, and behaviors of objects. An object, on the other hand, is an instance of a class, representing a specific occurrence or realization of that class.*

Each object created from a class has its own unique set of data values for the attributes defined in the class. These data values represent the state of the object. Additionally, objects possess the behaviors or methods defined in the class, which determine how they can interact and manipulate their data.

Objects can also be associated with each other, forming relationships and interactions within a program. For example, in a social networking application ([Figure 2.1](#)), User objects can be associated with other User objects through friendship connections, and Post objects can be associated with User objects as the creators of the posts. By creating multiple instances of a class (i.e., multiple objects), each object can have its own specific data values and can behave independently. This allows for the modeling of complex systems where different objects interact and collaborate with each other.

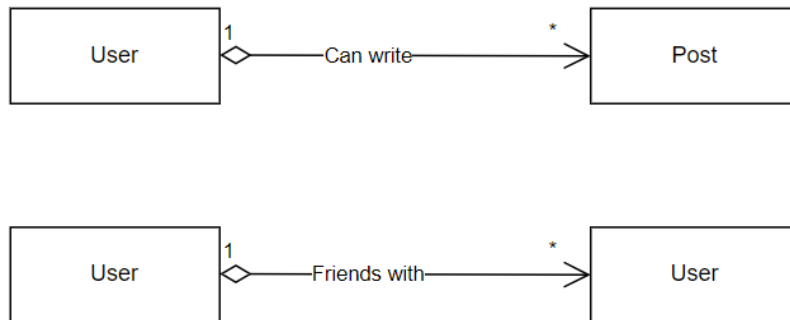


Figure 2-0-1

1.5. Programmatic representation of Class

Following is the program code in Python for User Class for the Social Media example discussed above:

```

class User:
    """
    Represents a user in the social networking application.

    Attributes:
        username (str): The username of the user.
        email (str): The email address of the user.
        password (str): The password of the user.
    """

    def __init__(self, username, email, password) -> None:
        """
        Represents a user in the social networking application.

        Attributes:
            username (str): The username of the user.
            email (str): The email address of the user.
            password (str): The password of the user.
        """
        self.username = username
        self.email = email
        self.password = password

    def add_friend(self) -> None:
        """
        Adds another User object as a friend.
        """
        # Write your own code
        pass

    def create_post(self) -> None:
        """
        Adds another User object as a friend.
        """
        # Write youe own code
        pass
  
```

In the example code above, we have created a User class with **attributes** such as username, email, and password. We have also identified two behaviors of this class: create post and add friend. Behaviors are actions that can occur on an object. In programming sense, behaviors are also termed as **methods**. Methods accept parameters and attributes of the same class or actual object instances of a class can be passed to a method as an **argument**. Methods can also **return values**.

In the fascinating world of **Object-Oriented Programming (OOP)**, we create virtual realms, intricate systems of entities with their own unique attributes, capabilities, and interconnectedness. This incredible concept finds its inspiration in the tapestry of human behavior that we encounter in our daily lives.

Let us imagine a bustling school, a microcosm teeming with life and purpose. Within this educational system, we encounter a vibrant tapestry of characters (entities): the wise and nurturing teachers, the authoritative headmaster, the eager students, the diligent clerks, the helpful associates, and the knowledgeable librarians. Each of these individuals assumes a distinct role and carries specific responsibilities within the grand scheme of the school.

*Picture, for instance, a section of the library reserved exclusively for students above the 10th grade(authorization). School security mandates authentication through identity cards before granting access to the school premises. Delving deeper, we uncover a clever **abstraction** that shields junior teachers in the hierarchy from the intricate process of question paper creation for exams. This intricate task is reserved for a select few, carefully chosen for their expertise. By segregating teachers based on their academic qualifications, we distinguish junior teachers from their esteemed counterparts who hold a distinguished Ph.D., thus granting them the coveted title of senior teachers or Professors(**hierarchy**).*

*As we explore further, we discover the meticulous orchestration of the school's financial affairs. Here, the entities seeking financial assistance must engage with a diligent clerk who acts as the conduit between them and the enigmatic realm of the account's office (**encapsulation**).*

*Resourceful helper staff members lend their skills and versatility to fulfill multiple utility tasks across the daily functioning of the school (**polymorphism**).*

1.6. Important Concepts

1.6.1. Abstraction

Abstraction is used to manage complexities. A clerk while typing on a computer screen is not bothered by the internal workings of a computer. He will use a mouse, keyboard, and computer screen to perform his tasks on a Software. On the other hand, a hardware engineering would be dealing with different physical parts of computer such as CPU, RAM, Hard Drives, etc. and connecting them all together to make it work. Here, clerk and hardware engineer both work at different level of abstraction.



Abstraction means dealing with the level of detail that is most appropriate to a given task.

Hierarchical Classification is an effective way to manage abstraction. It helps us to break complex systems in more manageable pieces. We consider a computer as a single system, but internally it is divided into many sub systems. For dealing with physical parts, we have a hardware subsystem, which in turn deals with objects like CPU, Disk, CD, Monitor, Keyboard, Mouse, etc. These objects have their own workings and are made of many such sub systems.

This concept applies to Object Oriented Programming as well. Abstraction is the process of encapsulating information with a public interface. In programming, abstraction is typically achieved

through the use of abstractions such as classes, interfaces, and functions. These abstractions define a set of behaviors or functionalities while hiding the internal workings or implementation details.

A class can be considered a layer of abstraction, which provides an interface to create objects and implement certain behaviors, yet information hiding can be done via **private members** of the class. The class provides a clear interface (methods and properties) for interacting with the object, while the internal implementation details are hidden from the outside world. This allows other parts of the program to use the class without needing to know how it is implemented internally, promoting modularity and reusability.



Abstraction allows programmers to work with high-level concepts and constructs that are closer to human understanding, rather than dealing with low-level implementation details. It provides a way to manage complexity, increase reusability, and improve the overall design and maintainability of software systems.

1.6.2. Encapsulation

Encapsulation, as the name suggests, refers to process of information hiding, in our case, it would be internal implementation of a class. However, it has broader aspect than just information hiding.



Encapsulation is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse, by controlling access to them to ensure data integrity and provide a well-defined interface to interact with the object.

In encapsulation, the internal state of an object is hidden from the outside world, and only selected methods, known as accessors and mutators (getters and setters), are provided to manipulate the object's data. This helps in preventing direct access to the object's data and ensures that the object's state is accessed and modified in a controlled manner.

Let us look at a sample Java Program which tries to create a Bank account system in programming world.

```
import java.util.List;

/**
 * A Bank account class
 */
public class BankAccount {
    private int accountNumber;
    private String accountHolderName;
    private double balance;
    private List<Transaction> transactionHistory;

    public int getAccountNumber() {
        return accountNumber;
    }
    public void setAccountNumber(int accountNumber) {
        this.accountNumber = accountNumber;
    }
    public String getAccountHolderName() {
        return accountHolderName;
    }
    public void setAccountHolderName(String accountHolderName) {
        this.accountHolderName = accountHolderName;
    }
    public void setBalance(double balance) {
        this.balance = balance;
    }
}
```

```

// Constructor and other methods...

/**
 * Method for depositing amount to account
 * @param amount
 */
public void deposit(double amount) {
    // Logic for depositing amount to the account
    // Update balance and transaction history
}

/**
 * Method to withdraw money from account
 * @param amount
 */
public void withdraw(double amount) {
    // Logic for withdrawing amount from the account
    // Update balance and transaction history
}

/**
 * Method to fetch balance of the account.
 * @return
 */
public double getBalance() {
    return balance;
}
}

```

In this example, the *BankAccount* class encapsulates the account details and provides public methods (*deposit*, *withdraw*, *getBalance*) to interact with the account. The internal data fields are **private**, preventing direct access from outside the class. Users or other parts of the banking system can only access and modify the account's data through the defined **public** methods, ensuring data integrity and security. Different programming languages may have different syntax, but underlying concept remains the same.



Encapsulation enhances code maintainability by localizing changes. If the internal representation of an object changes, only the class itself needs to be modified while the external code remains unaffected. This reduces the ripple effect of changes throughout the codebase, making it easier to manage and maintain. Also, these objects can be easily used in different parts of the codebase or in other projects, as they provide a well-defined interface for interaction, thus promoting code reusability.

1.6.3. Inheritance

To inherit or acquire is Inheritance. In the object-oriented world, classes (**subclass or child class**) inherit or acquire certain properties or behaviors from other class (**superclass or parent class**). Hierarchical relationships between classes is formed where a subclass can inherit and extend or override the characteristics of a superclass.



Inheritance establishes an "IS-A" relationship between the superclass and subclass. This means that a subclass is a specialized type of the superclass.

Let us look at it from the Banking context via a simple Java Program:

```

/**
 * Represents a bank account.
 */
public class SavingsAccount extends BankAccount {
    private double interestRate;
}

```

```

    /**
     * Constructs a SavingsAccount object with the specified account number, account holder
    name,
     * initial balance, and interest rate.
     *
     * @param accountNumber    the account number
     * @param accountHolderName the account holder's name
     * @param balance          the initial balance
     * @param interestRate     the interest rate for the savings account
     */
    public SavingsAccount(int accountNumber, String accountHolderName, double balance, double
    interestRate) {
        super(accountNumber, accountHolderName, balance);
        this.interestRate = interestRate;
    }

    /**
     * Applies interest to the savings account by calculating the interest based on the
    current balance
     * and adding it to the account.
     */
    public void applyInterest() {
        double interest = getBalance() * interestRate;
        deposit(interest);
    }
}

```

Here, Savings Account "IS-A" Bank Account. We create a *SavingsAccount* object, deposit some funds using the inherited *deposit()* method from the *BankAccount* class, and then apply interest using the *applyInterest()* method specific to the *SavingsAccount* class. Thus, we have extended and customized the functionality of the *BankAccount* class to create a specialized type of account (*SavingsAccount*) that includes additional behavior specific to savings accounts.

Let us extend this example by considering behavior of a Checking Account where withdrawal can be done from the balance amount and if there is not sufficient fund then an overdraft limit can be availed in case of emergency. So, the withdrawal method of checking account is different from the basic withdrawal method followed by Bank accounts by default.

Here is the code example of Checking Account where withdrawal functionality can be overridden, and its own custom functionality can be implemented.

```

    /**
     * Represents a checking bank account.
     */
    public class CheckingAccount extends BankAccount {
        private double overdraftLimit;

        /**
         * Constructs a CheckingAccount object with the specified account    number, account
        holder name,
         * initial balance, and overdraft limit.
         *
         * @param accountNumber    the account number
         * @param accountHolderName the account holder's name
         * @param balance          the initial balance
         * @param overdraftLimit    the overdraft limit for the checking account
         */
        public CheckingAccount(int accountNumber, String accountHolderName, double balance, double
        overdraftLimit) {
            super(accountNumber, accountHolderName, balance);
            this.overdraftLimit = overdraftLimit;
        }
    }

```



```

    /**
     * Withdraws the specified amount from the checking account, allowing overdraft up to the
     * overdraft limit.
     * If the withdrawal exceeds the balance and the available overdraft, an error message is
     * displayed.
     *
     * @param amount the amount to withdraw
     */
    public void withdraw(double amount) {
        double balance = getBalance();
        if (balance + overdraftLimit >= amount) {
            balance -= amount;
            setBalance(balance);
        } else {
            System.out.println("Insufficient funds");
        }
    }
}

```

1.6.4. Polymorphism

A Lamborghini or a Mustang IS-A car. Car can be used a generic interface to describe properties (attributes) such as wheel, steering, headlight, etc. and behaviors (methods) such as driving, honking, etc. of both Lamborghini and Mustang.

Similarly, A Savings Account or a Checking account IS-A Bank Account. However, if a Bank Account is savings account, it will not allow you to withdraw money if you have zero balance but for a checking account, you can withdraw availing an overdraft limit.



Polymorphism is a feature in object-oriented programming that allows objects of different classes to be treated as objects of a common superclass or interface. It enables a single interface to be used for a general class of actions, providing flexibility and code reuse.

Let us finalize the code for our Banking Example to conclude:

```

public class BankingExample {
    /**
     * @param args
     */
    public static void main(String[] args) {
        BankAccount account1 = new SavingsAccount(123456, "John Doe", 1000.0, 0.05);
        BankAccount account2 = new CheckingAccount(987654, "Jane Smith", 2000.0, 1000.0);

        account1.deposit(500.0);
        account1.withdraw(200.0);

        account2.deposit(1000.0);
        account2.withdraw(3000.0);

        System.out.println("Account 1 balance: " + account1.getBalance());
        System.out.println("Account 2 balance: " + account2.getBalance());
    }
}

```

In the *BankingExample* class, we create instances of *SavingsAccount* and *CheckingAccount* but store them in variables of type *BankAccount*. Polymorphism allows us to treat instances of *SavingsAccount* and *CheckingAccount* as objects of the *BankAccount* superclass. This flexibility enables us to use a single interface (*BankAccount*) to perform operations on different types of accounts.



Caution

If we instantiate a *SavingsAccount* object using the *BankAccount* class reference, we will only have access to the methods and members that are defined in the *BankAccount* class. Although we can't directly access the specific methods of subclasses when using the superclass reference, polymorphism allows us to override and provide different implementations of methods in the subclasses. We shall look at these design problems in coming chapters.

1.7. Summary: Encapsulation, Inheritance and Polymorphism

In Object-Oriented programming world, Encapsulation, inheritance, and polymorphism combine to produce a programming environment that supports the development of far more robust and scalable programs. Some of the key insights we have covered in this chapter are as follows:

- **Robust and Scalable Programs:** Polymorphism, encapsulation, and inheritance contribute to the development of robust and scalable programs. By using inheritance, we can create a hierarchy of classes that promotes code reuse and organization. This allows us to build upon existing code and extend functionality without starting from scratch. Additionally, encapsulation ensures that the internal implementation details of a class are hidden, reducing dependencies, and making it easier to modify and maintain code over time.
- **Code Reusability:** Inheritance enables code reuse by allowing subclasses to inherit properties and methods from their superclass. This saves time and effort by avoiding the need to rewrite common functionality. Through proper design, we can create a hierarchy of classes where subclasses inherit and extend the behavior of their parent classes. This code reuse enhances productivity and maintainability.
- **Migration and Compatibility:** Encapsulation plays a crucial role in maintaining code compatibility. By encapsulating the internal implementation details of a class, we can modify or improve that implementation without affecting the code that depends on the public interface of the class. This allows for gradual migration and evolution of the software, minimizing the risk of breaking existing code.
- **Clean and Readable Code:** Polymorphism allows us to write clean, sensible, and readable code. By designing our classes and interfaces with polymorphism in mind, we can create code that is more flexible and adaptable. Polymorphism enables us to write code that operates on a general interface or superclass, making it easier to understand and maintain. It also promotes code that follows the principles of abstraction and separation of concerns.
- **Resilient Code:** Polymorphism aids in the creation of resilient code. With polymorphism, you can write code that can handle various types of objects without explicitly checking their specific types. This leads to more flexible and adaptable code that can handle different scenarios and changes in requirements.

In the next chapter, we shall discuss some of the design patterns used in Object oriented world to provide simple and elegant solutions to specific problems. By studying design patterns, we gain a deeper understanding of various problem domains and their corresponding design considerations.