Why Do We Need Classes and Objects?

Classes and objects are used in programming to organize and structure code in a way that makes it easier to manage and reuse.

The Need for Structure and Organization

Let's consider a scenario without classes and objects. Suppose we want to create a program to manage bank accounts for multiple customers. We could use separate variables for each customer's account number, account holder name, and balance. For example:

In [1]:
```python
account1_number = "1234567890"
account1_holder = "John"
account1_balance = 5000

account2_number = "0987654321"
account2_holder = "Jane"
account2_balance = 10000
```

While this approach may work for a small number of accounts, it quickly becomes unwieldy and difficult to manage as the number of accounts grows. It becomes challenging to keep track of all the variables and their relationships.

Introducing Classes and Objects

Here's where classes and objects come into play. By using classes, we can create a blueprint for bank accounts and define their properties and behaviors. Each individual account becomes an object, an instance of the class.

With classes and objects, we can rewrite the above code as follows:

In [45]:
```
class BankAccount:

    def __init__(self, account_number, account_holder, balance):
        self.account_number = account_number
        self.account_holder = account_holder
        self.balance = balance




# Create instances of BankAccount for John and Jane
customer_1 = BankAccount("1234567890", "John", 5000)
customer_2 = BankAccount("0987654321", "Jane", 10000)
# customer_1 = BankAccount()
# account1.test()
```

In [46]:
```
# customer1 = BankAccount("","",5000)
# customer_2.intrest_rate
```

In [ ]:
```

```

By using the BankAccount class, we have a structured way to represent and manage bank accounts. Each account object (e.g., account1 and account2) has its own unique set of attributes (e.g., account_number, account_holder, and balance).

- class BankAccount: This line defines a new class called BankAccount. Think of a class as a blueprint or template that describes the properties (attributes) and behaviors (methods) that objects of that class will have.
- def __init__(self, account_number, account_holder, balance): This line defines a special method called __init__. It is a constructor method, which gets called automatically when a new object of the class is created. The self parameter refers to the object being created, and the other parameters (account_number, account_holder, and balance) are the initial values we provide when creating an object.
- self.account_number = account_number, self.account_holder = account_holder, and self.balance = balance: These lines assign the values passed as arguments to the corresponding attributes (account_number, account_holder, and balance) of the object being created. The self keyword refers to the current object, and it allows us to store these values as attributes of that object.

NOTE:

The **init** method is not compulsory to write in a class, but it is commonly used to initialize the object's attributes with some initial values. It provides a convenient way to set up the object's state when it is created.

In [52]:
```
# Conclusion : Think of a class as a blueprint or template that describes the
class BankAccount:
    def __init__(self, account_number, account_holder, balance):
        self.account_number = account_number
        self.account_holder = account_holder
        self.balance = balance

    def deposit(self, amount):
        self.balance += amount
        print("Deposited", amount, "into the account.")

    def withdraw(self, amount):
        if self.balance >= amount:
            self.balance -= amount
            print("Withdrawn", amount, "from the account.")
        else:
            print("Insufficient funds. Cannot withdraw", amount, "from the ac

    def display_balance(self):
        print("Account Holder:", self.account_holder)
        print("Account Number:", self.account_number)
        print("Balance:", self.balance)
```

In [53]:
```
my_account = BankAccount("1234567890", "John Doe", 5000)
my_account.withdraw(122)
# BankAccount.withdraw(122)
```

```
Withdrawn 122 from the account.
```

In [54]:
```
my_account = BankAccount("1234567890", "John Doe", 5000)
my_account.withdraw(122)
my_account.display_balance()
```

```
Withdrawn 122 from the account.
Account Holder: John Doe
Account Number: 1234567890
Balance: 4878
```

In [56]:
```
# lst =[1,2,3,4]
# type(my_account)
```

Out[56]:
```
__main__.BankAccount
```

In [ ]:
```
lst.append()
```

In [ ]:

In [57]:
```python
# Create a bank account object
my_account = BankAccount("1234567890", "John Doe", 5000)

# Perform operations on the account
my_account.display_balance()  # Display the account balance
my_account.deposit(2000)      # Deposit 2000 into the account
my_account.withdraw(3000)     # Withdraw 3000 from the account
my_account.display_balance()  # Display the updated account balance
```

```
Account Holder: John Doe
Account Number: 1234567890
Balance: 5000
Deposited 2000 into the account.
Withdrawn 3000 from the account.
Account Holder: John Doe
Account Number: 1234567890
Balance: 4000
```

Methods

In object-oriented programming, methods are functions defined within a class that perform specific actions or provide certain functionalities related to the objects of that class. Methods are used to encapsulate behavior and allow objects to interact with and manipulate their own data (attributes).

Types Of Methods

- Instance Methods : Instance methods are the most common type of methods in a class. They are defined within a class and operate on individual instances (objects) of that class. These methods typically have the self parameter as the first parameter, which allows the method to access and manipulate the object's attributes. The deposit, withdraw, and display_balance methods in the BankAccount example are instance methods.

In [58]:
```
class BankAccount:
    def __init__(self, account_number, account_holder, balance):
        self.account_number = account_number
        self.account_holder = account_holder
        self.balance = balance

    def deposit(self, amount):
        self.balance += amount
        print("Deposited", amount, "into the account.")

    def withdraw(self, amount):
        if self.balance >= amount:
            self.balance -= amount
            print("Withdrawn", amount, "from the account.")
        else:
            print("Insufficient funds. Cannot withdraw", amount, "from the ac

    def display_balance(self):
        print("Account Holder:", self.account_holder)
        print("Account Number:", self.account_number)
        print("Balance:", self.balance)

    def transfer_funds(self, destination_account, amount):
        if self.balance >= amount:
            self.balance -= amount
            destination_account.balance += amount
            print("Transferred", amount, "to account", destination_account.ac
        else:
            print("Insufficient funds. Cannot transfer", amount, "to account"
```

In [59]:
```
my_account = BankAccount("1234567890", "John Doe", 5000)
friend_account = BankAccount("0987654321", "Jane Smith", 2000)
```

In [60]:
```
my_account.transfer_funds(friend_account,200)
# friend_account.transfer_funds()
```

```
Transferred 200 to account 0987654321
```

In [61]:
```
# Create bank account objects
my_account = BankAccount("1234567890", "John Doe", 5000)
friend_account = BankAccount("0987654321", "Jane Smith", 2000)

# Perform operations using instance methods
my_account.display_balance()                   # Display my account balanc
friend_account.display_balance()               # Display friend's account 
my_account.transfer_funds(friend_account, 3000)  # Transfer 3000 to friend'
my_account.display_balance()                   # Display updated balance o
friend_account.display_balance()               # Display updated balance o
```

```
Account Holder: John Doe
Account Number: 1234567890
Balance: 5000
Account Holder: Jane Smith
Account Number: 0987654321
Balance: 2000
Transferred 3000 to account 0987654321
Account Holder: John Doe
Account Number: 1234567890
Balance: 2000
Account Holder: Jane Smith
Account Number: 0987654321
Balance: 5000
```

So in above example, all function have self as first parameter then I can say all are instance method ?:

Yes, exactly! In the BankAccount example we discussed earlier, all the functions (deposit, withdraw, display_balance, and transfer_funds) have self as their first parameter. Therefore, they are all considered instance methods.

So, in the context of the BankAccount class, all the methods defined (deposit, withdraw, display_balance, and transfer_funds) are instance methods. They operate on the individual instances of the BankAccount class and can access and modify the attributes (account_number, account_holder, and balance) of those instances using the self parameter.

- **Class Methods:** Class methods are methods that are bound to the class rather than an instance of the class. They are defined using the @classmethod decorator and take the class itself as the first parameter (usually named cls). Class methods can access and modify class-level attributes and perform operations that are specific to the class as a whole. They are often used for alternative constructors, utility methods, or methods that operate on class-level data. Here's an example:

In [62]:
```
class BankAccount:
    interest_rate = 0.05

    def __init__(self, account_number, account_holder, balance):
        self.account_number = account_number
        self.account_holder = account_holder
        self.balance = balance

    def deposit(self, amount):
        self.balance += amount
        print("Deposited", amount, "into the account.")

    def withdraw(self, amount):
        if self.balance >= amount:
            self.balance -= amount
            print("Withdrawn", amount, "from the account.")
        else:
            print("Insufficient funds. Cannot withdraw", amount, "from the ac

    def display_balance(self):
        print("Account Holder:", self.account_holder)
        print("Account Number:", self.account_number)
        print("Balance:", self.balance)

    @classmethod
    def change_interest_rate(cls, new_rate):
        cls.interest_rate = new_rate
        print("Interest rate updated to", new_rate)


# Create bank account objects
my_account = BankAccount("1234567890", "John Doe", 5000)
friend_account = BankAccount("0987654321", "Jane Smith", 2000)

# Display initial interest rate
print("Initial Interest Rate:", BankAccount.interest_rate)

# Change the interest rate using the class method
BankAccount.change_interest_rate(0.07)

# Display updated interest rate
print("Updated Interest Rate:", BankAccount.interest_rate)
```

```
Initial Interest Rate: 0.05
Interest rate updated to 0.07
Updated Interest Rate: 0.07
```

To summarize:

- Instance methods are used to work with individual instances and can access instance attributes through the self parameter.
- Class methods are used to work with the class itself and can access class-level attributes through the cls parameter.

- **Static Methods:** Static methods are methods that do not have access to the instance (self) or the class (cls) and behave like regular functions within the class. They are defined using the @staticmethod decorator and are mainly used for utility functions that don't depend on instance or class-specific data. Static methods can be called on the class or an instance of the class. Here's an example:

In [78]:
```
class BankAccount:
    interest_rate = 0.05

    def __init__(self, account_number, account_holder, balance):
        self.account_number = account_number
        self.account_holder = account_holder
        self.balance = balance

    def deposit(self, amount):
        self.balance += amount
        print("Deposited", amount, "into the account.")

    def withdraw(self, amount):
        if self.balance >= amount:
            self.balance -= amount
            print("Withdrawn", amount, "from the account.")
        else:
            print("Insufficient funds. Cannot withdraw", amount, "from the ac

    def display_balance(self):
        print("Account Holder:", self.account_holder)
        print("Account Number:", self.account_number)
        print("Balance:", self.balance)

    @classmethod
    def change_interest_rate(cls, new_rate):
        cls.interest_rate = new_rate
        print("Interest rate updated to", new_rate)

    @staticmethod
    def is_balance_positive(balance):
        return balance > 0
```

In [79]:
```
c1 = BankAccount('123A','Rahul',1500)
```

In [84]:
```
# BankAccount.is_balance_positive(100)
```

In [ ]:

```

```

In [ ]:

```

```

To use the static method, you can call it directly on the class itself, without creating an instance of the class. Here's an example:

In [65]:

```python
print(BankAccount.is_balance_positive(1000))   # Output: True
print(BankAccount.is_balance_positive(-500))   # Output: False
```

```
True
False
```

Inheritance

Inheritance is a fundamental concept in object-oriented programming that allows you to create new classes based on existing classes. It enables the creation of a hierarchy of classes, where the child classes (known as subclasses or derived classes) inherit the attributes and behaviors of the parent class (known as the superclass or base class).

In [88]:

```python
class SavingsAccount(BankAccount):
    def __init__(self, account_number, account_holder, balance):
        super().__init__(account_number, account_holder, balance)
        self.interest_rate = 0.1

    def add_interest(self):
        interest_amount = self.balance * self.interest_rate
        self.balance += interest_amount
        print("Added interest of", interest_amount, "to the account.")
```

In [91]:

```python
Alice = SavingsAccount("9876543210", "Alice Smith", 10000)
```

In [92]:

```python
Alice.add_interest()
```

```
Added interest of 1000.0 to the account.
```

In [93]:

```python
Alice.balance
```

Out[93]:

```
11000.0
```

In [94]:

```python
Alice.add_interest()
```

```
Added interest of 1100.0 to the account.
```

In [95]:
```
Alice.balance
```

Out[95]:
```
12100.0
```

In [ ]:
```
```

- The SavingsAccount class is defined with BankAccount specified in parentheses after the class name. This indicates that SavingsAccount is a subclass of BankAccount and inherits from it.
- In the __init__ method of SavingsAccount, we use the super() function to call the __init__ method of the parent class (BankAccount). This initializes the inherited attributes of the BankAccount class.
- The SavingsAccount class introduces a new attribute called interest_rate specific to savings accounts.
- The add_interest method calculates the interest amount based on the balance and interest rate, adds it to the account balance, and prints a message.

In [96]:
```
savings_account = SavingsAccount("9876543210", "Alice Smith", 10000)
```

In [87]:
```
# savings_account.
```

In [97]:
```
savings_account = SavingsAccount("9876543210", "Alice Smith", 10000)
savings_account.deposit(500)
savings_account.display_balance()
savings_account.add_interest()
savings_account.display_balance()
```

```
Deposited 500 into the account.
Account Holder: Alice Smith
Account Number: 9876543210
Balance: 10500
Added interest of 1050.0 to the account.
Account Holder: Alice Smith
Account Number: 9876543210
Balance: 11550.0
```

Conclusion:

Inheritance allows for code reuse, promotes modularity, and facilitates the creation of specialized classes based on a common base. It helps in organizing classes into a hierarchical structure, where subclasses inherit and build upon the characteristics of their parent classes.

Polymorphism

Polymorphism is a key concept in object-oriented programming that allows objects of different classes to be treated as objects of a common superclass. It enables the same method or function to be called on different objects, even if they belong to different classes, as long as they share a common interface or base class.

In [104]:
```
class BankAccount:
    def __init__(self, account_number, account_holder, balance):
        self.account_number = account_number
        self.account_holder = account_holder
        self.balance = balance

    def display_balance(self):
        print("Account Holder:", self.account_holder)
        print("Account Number:", self.account_number)
        print("Balance:", self.balance)

    def withdraw(self, amount):
        if self.balance >= amount:
            self.balance -= amount
            print("Withdrawn", amount, "from the account.")
        else:
            print("Insufficient funds. Cannot withdraw", amount, "from the ac


class SavingsAccount(BankAccount):
    def __init__(self, account_number, account_holder, balance):
        super().__init__(account_number, account_holder, balance)
        self.interest_rate = 0.1

    def add_interest(self):
        interest_amount = self.balance * self.interest_rate
        self.balance += interest_amount
        print("Added interest of", interest_amount, "to the account.")


class CheckingAccount(BankAccount):
    def __init__(self, account_number, account_holder, balance, overdraft_lim
        super().__init__(account_number, account_holder, balance)
        self.overdraft_limit = overdraft_limit

    def withdraw(self, amount):
        if self.balance + self.overdraft_limit >= amount:
            self.balance -= amount
            print("Withdrawn", amount, "from the account.")
        else:
            print("Insufficient funds. Cannot withdraw", amount, "from the ac

# def perform_withdrawal(account, amount):
#     account.withdraw(amount)

# # Create instances of different account types
# savings_account = SavingsAccount("9876543210", "Alice Smith", 10000)
# checking_account = CheckingAccount("1234567890", "Bob Johnson", 5000, 1000)

# # Call the common method perform_withdrawal on different account objects
# perform_withdrawal(savings_account, 5000)    # Output: Withdrawn 5000 from
# perform_withdrawal(checking_account, 8000)   # Output: Withdrawn 8000 from
```

In [105]:
```
rahul = CheckingAccount('123A','Rahul',5000,1000)
```

In [106]:
```
rahul.withdraw(5500)
```

```
Withdrawn 5500 from the account.
```

In [107]:
```
rahul.balance
```

Out[107]:
```
-500
```

Data abstraction

Data abstraction is a fundamental concept in object-oriented programming that allows us to focus on essential attributes and behaviors of an object while hiding the unnecessary details. It enables us to create complex systems by breaking them down into manageable, modular components.

we have a BankAccount class that represents a bank account object. The class has attributes such as account_number, account_holder, and balance, which store the relevant data associated with a bank account.

The class also defines methods like display_balance and withdraw to interact with the bank account object. The display_balance method displays the account holder's information and current balance. The withdraw method allows withdrawing a specified amount from the account, considering the available balance.

By defining the class and its methods, we encapsulate the internal details of a bank account. The user of the class doesn't need to know how the account's data is stored or how the withdrawal process is implemented. They only interact with the bank account object through its interface (methods), which provides the necessary abstractions to work with the account.

Encapsulation

Encapsulation refers to the bundling of data (attributes) and methods (behavior) within a class, where the data is typically hidden or protected from direct access from outside the class. This encapsulation provides several benefits, such as data protection, abstraction, and improved code maintainability.

In [117]:
```
class BankAccount:
    def __init__(self, account_number, account_holder, balance):
        self.account_number = account_number
        self.account_holder = account_holder
        self.__balance = balance

    def display_balance(self):
        print("Account Holder:", self.account_holder)
        print("Account Number:", self.account_number)
        print("Balance:", self.__balance)

    def withdraw(self, amount):
        if self.__balance >= amount:
            self.__balance -= amount
            print("Withdrawn", amount, "from the account.")
        else:
            print("Insufficient funds. Cannot withdraw", amount, "from the ac
```

Encapsulation refers to the bundling of data (attributes) and methods (behavior) within a class, where the data is typically hidden or protected from direct access from outside the class. This encapsulation provides several benefits, such as data protection, abstraction, and improved code maintainability.

In our example, we have made the __balance attribute private by using the double underscore prefix. This indicates that the attribute is intended to be accessed only within the class itself, and it should not be directly accessed or modified from outside the class.

The display_balance method provides an interface to access the balance information indirectly. It can access the private __balance attribute and display it to the user.

The withdraw method also operates on the private __balance attribute. It checks if the withdrawal amount is within the available balance and updates the balance accordingly.

By encapsulating the __balance attribute and providing controlled access through methods, we ensure that the balance can only be modified through the defined interface of the class. This encapsulation helps in maintaining data integrity and prevents unwanted modifications or direct access to the internal state of the object.

In [121]:
```
# Create instances of BankAccount
account1 = BankAccount("123456789", "John Doe", 5000)
account2 = BankAccount("987654321", "Jane Smith", 10000)

account1.display_balance()
```

```
Account Holder: John Doe
Account Number: 123456789
Balance: 5000
```

In [122]:
```
# Access and display the account balance
account1.display_balance()  # Output: Account Holder: John Doe Account Number

# Withdraw funds from the account
account1.withdraw(2000)  # Output: Withdrawn 2000 from the account.

# Display the updated account balance
account1.display_balance()  # Output: Account Holder: John Doe Account Number
```

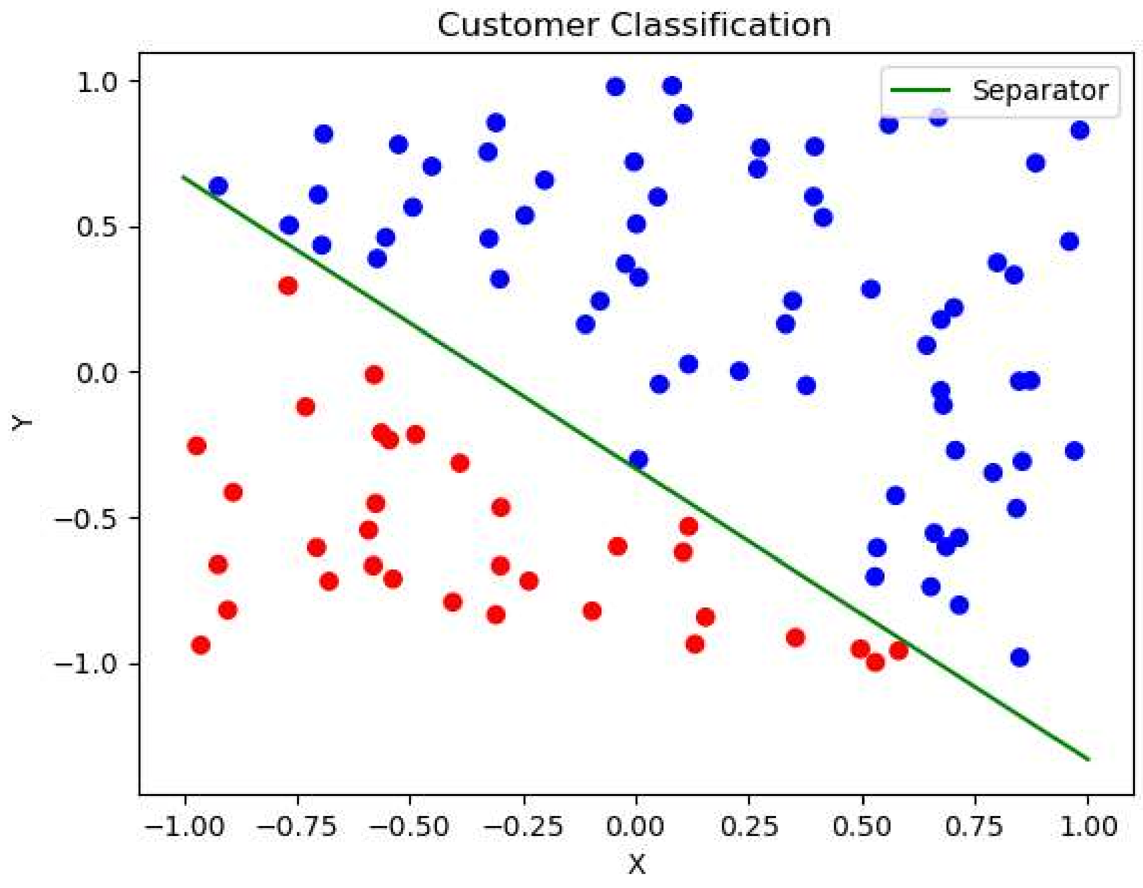
```
Account Holder: John Doe
Account Number: 123456789
Balance: 5000
Withdrawn 2000 from the account.
Account Holder: John Doe
Account Number: 123456789
Balance: 3000
```

In [111]:
```
# def predict(x):
#     output = 3*x+4
#     return output


# var1=3131
# var
```

Misc Example:

In [112]:
```python
import random
import matplotlib.pyplot as plt

class CustomerClassifier:
    def __init__(self, weights, bias):
        self.weights = weights
        self.bias = bias

    def predict(self, features):
        activation = sum(w * x for w, x in zip(self.weights, features)) + sel
        return 1 if activation >= 0 else 0

    def generate_random_points(self, num_points):
        data_points = []
        for _ in range(num_points):
            x = random.uniform(-1, 1)
            y = random.uniform(-1, 1)
            data_points.append((x, y))
        return data_points

    def plot_points(self, points):
        colors = ["red" if self.predict(p) == 1 else "blue" for p in points]
        x = [p[0] for p in points]
        y = [p[1] for p in points]
        plt.scatter(x, y, color=colors)

        # Plot the line
        line_x = [-1, 1]
        line_y = [-(self.weights[0]*x + self.bias)/self.weights[1] for x in l
        plt.plot(line_x, line_y, color="green", label="Separator")

        plt.xlabel("X")
        plt.ylabel("Y")
        plt.title("Customer Classification")
        plt.legend()
        plt.show()
```

In [113]:
```
# Create a customer classifier object
weights = [-0.6, -0.6]
bias = -0.2
classifier = CustomerClassifier(weights, bias)

# Generate random data points
num_points = 100
points = classifier.generate_random_points(num_points)

# Plot the points
classifier.plot_points(points)
```



In [124]:

```python
from datetime import datetime
class MarketingCampaign:
    def __init__(self, campaign_name, start_date, end_date):
        self.campaign_name = campaign_name
        self.start_date = start_date
        self.end_date = end_date
        self.clicks = []
        self.conversions = []

    def add_click(self, click_date):
        self.clicks.append(click_date)

    def add_conversion(self, conversion_date):
        self.conversions.append(conversion_date)

    def get_clicks(self):
        return len(self.clicks)

    def get_conversions(self):
        return len(self.conversions)

    def get_conversion_rate(self):
        if len(self.clicks) > 0:
            return len(self.conversions) / len(self.clicks)
        else:
            return 0.0

    def get_campaign_duration(self):
        start = datetime.strptime(self.start_date, "%Y-%m-%d")
        end = datetime.strptime(self.end_date, "%Y-%m-%d")
        return (end - start).days


class EmailCampaign(MarketingCampaign):
    def __init__(self, campaign_name, start_date, end_date, email_subject):
        super().__init__(campaign_name, start_date, end_date)
        self.email_subject = email_subject

    def send_email(self):
        print(f"Sending email with subject: {self.email_subject}")


class SocialMediaCampaign(MarketingCampaign):
    def __init__(self, campaign_name, start_date, end_date, platform):
        super().__init__(campaign_name, start_date, end_date)
        self.platform = platform

    def create_post(self, content):
        print(f"Creating {self.platform} post with content: {content}")


# Example usage
campaign1 = EmailCampaign("Welcome Campaign", "2023-01-01", "2023-01-31", "We
campaign1.add_click("2023-01-02")
campaign1.add_click("2023-01-05")
campaign1.add_conversion("2023-01-05")
```

```
campaign1.add_conversion("2023-01-10")

campaign2 = SocialMediaCampaign("Facebook Campaign", "2023-02-01", "2023-02-2
campaign2.add_click("2023-02-05")
campaign2.add_click("2023-02-10")
campaign2.add_conversion("2023-02-10")

print("Campaign:", campaign1.campaign_name)
print("Clicks:", campaign1.get_clicks())
print("Conversions:", campaign1.get_conversions())
print("Conversion Rate:", campaign1.get_conversion_rate())
print("Campaign Duration (in days):", campaign1.get_campaign_duration())

print("\nCampaign:", campaign2.campaign_name)
print("Clicks:", campaign2.get_clicks())
print("Conversions:", campaign2.get_conversions())
print("Conversion Rate:", campaign2.get_conversion_rate())
print("Campaign Duration (in days):", campaign2.get_campaign_duration())

campaign1.send_email()  # Polymorphism in action
campaign2.create_post("Check out our new product!")  # Polymorphism in action
```

```
Campaign: Welcome Campaign
Clicks: 2
Conversions: 2
Conversion Rate: 1.0
Campaign Duration (in days): 30

Campaign: Facebook Campaign
Clicks: 2
Conversions: 1
Conversion Rate: 0.5
Campaign Duration (in days): 27
Sending email with subject: Welcome to our newsletter!
Creating Facebook post with content: Check out our new product!
```

In [ ]: