Chapter 10: Object-Oriented Programming

Solving a problem by creating objects is one of the most popular approaches in programming. This is called object-oriented programming (OOP). This concept focuses on using reusable code (DRY Principle).

Class

A class is a blueprint for creating objects. It defines a set of attributes and methods that the created objects will have.

Syntax:

```
class Employee:  # Class name is written in Pascal case
    # Methods & Variables
    pass
```

Object

An object is an instantiation of a class. When a class is defined, a template (information) is defined. Memory is allocated only after object instantiation. Objects of a given class can invoke the methods available to it without revealing the implementation details to the user, adhering to the principles of abstraction and encapsulation.

Modelling a Problem in OOP

When modelling a problem, we identify the following:

- **Noun → Class → Employee**
- **Adjective → Attributes → name, age, salary**
- **Verbs → Methods → getSalary(), increment()**

Class Attributes

An attribute that belongs to the class rather than a particular object.

Example:

```python
class Employee:
    company = "Google"  # Class attribute

shivam = Employee()  # Object instantiation
print(shivam.company)  # Output: Google

Employee.company = "YouTube"  # Changing class attribute
print(shivam.company)  # Output: YouTube
```

Instance Attributes

An attribute that belongs to the instance (object).

Example:

```python
shivam.name = "Shivam"
shivam.salary = "30k"  # Adding instance attribute

# When looking up for shivam.attribute, it checks for:
# 1) Is the attribute present in the object?
# 2) Is the attribute present in the class?
```

Self Parameter

`self` refers to the instance of the class. It is automatically passed with a function call from an object.

Example:

```python
class Employee:
    company = "Google"

    def getSalary(self):
        print("Salary is not there")

shivam = Employee()
shivam.getSalary()  # here self is shivam
# Equivalent to Employee.getSalary(shivam)
```

Static Method

Sometimes we need a function that does not use the self-parameter. We can define a static method using a decorator.

Example:

```python
class Employee:
    @staticmethod  # Decorator to mark greet as a static method
    def greet():
        print("Hello user")
```

init() Constructor

`__init__()` is a special method which is first run as soon as the object is created. It is also known as the constructor. It takes `self` argument and can also take further arguments.

Example:

```python
class Employee:
    def __init__(self, name):
        self.name = name

    def getSalary(self):
        print(f"{self.name}'s salary is not there")

shivam = Employee("Shivam")
shivam.getSalary()  # Output: Shivam's salary is not there
```

Step-by-Step Explanation of Constructors.

1. **Class Definition:**

```python
class Book:
```

Here, we define a class named `Book`. A class is a blueprint for creating objects (instances), and it can contain attributes (data) and methods (functions).

Constructor Method (`__init__`):

```python
def __init__(self, title, author, pages=0):
```

The `__init__` method is a special method called a constructor. It is automatically called when a new object of the class is created.
The constructor takes three parameters:

- `title`: The title of the book.
- `author`: The author of the book.

- pages: The number of pages in the book, with a default value of 0

```
self.title = title
self.author = author
self.pages = pages
```

`self` refers to the current instance of the class.

- `self.title = title`: Assigns the value of the `title` parameter to the `title` attribute of the instance.
- `self.author = author`: Assigns the value of the `author` parameter to the `author` attribute of the instance.
- `self.pages = pages`: Assigns the value of the `pages` parameter to the `pages` attribute of the instance.

3. Display Method:

```
def display(self):
```

This method is used to print the details of the book

```
print(f"Title: {self.title}, Author: {self.author}, Pages: {self.pages}")
```

The `print` function outputs a formatted string that includes the title, author, and pages of the book instance.

4. Creating an Object of the Book Class:

```
book1 = Book("Python Programming", "Shivam")
```

This line creates a new object (instance) of the `Book` class named `book1`.

- The `__init__` method is called with the arguments `"Python Programming"` and `"Shivam"`. Since the `pages` parameter is not provided, it uses the default value of `0`.

Displaying the Details of `book1`:

```
book1.display()
```

This line calls the `display` method on the `book1` object.

- It prints: `Title: Python Programming, Author: Shivam, Pages: 0`.

6. Creating Another Object of the Book Class:

```
book2 = Book("Advanced Python", "Shivam", 350)
```

This line creates another object of the `Book` class named `book2`.

- The `__init__` method is called with the arguments `"Advanced Python"`, `"Shivam"`, and `350`. The `pages` parameter is provided, so it takes the value `350`.

7. Displaying the Details of `book2`:

```
book2.display()
```

This line calls the `display` method on the `book2` object.
It prints: `Title: Advanced Python, Author: Shivam, Pages: 350`.

Summary

- The `Book` class is defined with a constructor method to initialize its attributes (`title`, `author`, and `pages`).
- The `display` method is used to print the details of a book instance.
- Two instances of the `Book` class (`book1` and `book2`) are created with different attributes.
- The `display` method is called on each instance to print their details.

This example demonstrates how to define a class with a constructor, create instances of the class, and call methods on those instances to perform actions**.**