***OBJECT ORIENTED PROGRAMMING (OOPS)***

In programming, an **instance of a class** is a specific object created from a class. Think of a class as a **blueprint** or a **template** that defines *the properties (data) and behaviours (methods) that an object will have*. An instance, on the other hand, is the actual, tangible object built from that blueprint.

**Analogy**

*Consider a blueprint for a house. 🏠 The blueprint itself isn't a house you can live in; it's just the plan. The houses you build from that blueprint are the* ***instances***. Each house is unique—it has its own address, color, and contents—but it follows the same basic structure defined by the blueprint.

**Key Concepts**

* **Class:** A user-defined data type that acts as a blueprint for creating objects. It encapsulates data for the object and methods to manipulate that data.
* **Object:** A real-world entity or a self-contained component in a program. It is a specific realization of a class.
* **Instantiation:** The process of creating an instance of a class. This is typically done using a constructor or a specific keyword like new in many programming languages.

**Example in Python**

Let's use a simple Python example to illustrate.

Python

# The 'Car' class is the blueprint

class Car:

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

self.make = make

self.model = model

def display\_info(self):

return f"This car is a {self.make} {self.model}."

# Creating instances of the 'Car' class

car1 = Car("Toyota", "Camry")

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

# Each variable (car1, car2) is a unique instance of the 'Car' class

print(car1.display\_info()) # Output: This car is a Toyota Camry.

print(car2.display\_info()) # Output: This car is a Honda Civic.

In this code, Car is the class. car1 and car2 are two different **instances** of the Car class. They both have a make and model property and a display\_info method, as defined by the class, but their specific data ("Toyota", "Camry" and "Honda", "Civic") is unique to each instance.

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of "objects," which can contain data and code. It's a way of structuring programs to make them more modular, reusable, and easier to manage. Here's a breakdown of its core concepts.

**Classes**

A **class** is a blueprint or a template for creating objects. It defines the properties (data) and behaviors (methods) that all objects of that class will have. Think of it as the design for a car. It specifies that every car has a color, a make, a model, and can perform actions like start() or stop().

Example:

In Python, a Car class might look like this:

Python

class Car:

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

self.make = make

self.model = model

self.color = color

def start\_engine(self):

print(f"The {self.color} {self.make} {self.model}'s engine is starting.")

**Objects**

An **object** is a specific instance of a class. It's a real-world entity created from the class blueprint. Each object has its own unique set of data, but it shares the same behaviors defined by the class. Following the car analogy, a specific Toyota Camry that's red is an object.

Example:

Using the Car class, we can create two distinct objects:

Python

car1 = Car("Toyota", "Camry", "Red")

car2 = Car("Honda", "Civic", "Blue")

car1.start\_engine() # Output: The Red Toyota Camry's engine is starting.

car2.start\_engine() # Output: The Blue Honda Civic's engine is starting.

Here, car1 and car2 are two separate objects, each with its own make, model, and color.

**Inheritance**

**Inheritance** is a mechanism that allows a new class (**subclass** or **child class**) to inherit properties and methods from an existing class (**superclass** or **parent class**). It promotes code reuse and establishes a hierarchical "is-a" relationship (e.g., a "Sedan is a Car"). The child class can extend or override the parent's functionality.

Example:

Let's create a Sedan subclass that inherits from the Car class.

Python

class Sedan(Car):

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

super().\_\_init\_\_(make, model, color) # Inherit parent's properties

self.num\_doors = num\_doors

def open\_trunk(self):

print("The trunk is open.")

sedan1 = Sedan("Honda", "Accord", "Silver", 4)

sedan1.start\_engine() # Inherited from Car class

sedan1.open\_trunk() # Unique to Sedan class

**Encapsulation**

**Encapsulation** is the bundling of data and the methods that operate on that data into a single unit (the class). It's the practice of hiding the internal state of an object from the outside world and only exposing a controlled interface. This is often achieved using access modifiers (private, public, etc.) to prevent direct, unauthorized access to an object's data.

Example:

Consider a BankAccount class. We should prevent direct access to the \_balance to ensure its integrity.

Python

class BankAccount:

def \_\_init\_\_(self, initial\_balance):

self.\_\_balance = initial\_balance # The '\_\_' makes it a private variable

def deposit(self, amount):

if amount > 0:

self.\_\_balance += amount

print(f"Deposited {amount}. New balance is {self.\_\_balance}.")

def get\_balance(self):

return self.\_\_balance

account = BankAccount(100)

# This will cause an error or warning because direct access is discouraged/prevented:

# print(account.\_\_balance)

account.deposit(50) # Correct way to modify the balance

print(account.get\_balance())

**Polymorphism**

**Polymorphism** means "many forms." In OOP, it's the ability of an object or method to take on many forms. This allows objects of different classes to be treated as objects of a common superclass. This is typically achieved through method overriding or overloading.

Example:

Let's use the Car and Sedan classes and a new Truck class. All three can have a get\_type() method, but each provides a unique implementation.

Python

class Truck(Car):

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

super().\_\_init\_\_(make, model, color)

def get\_type(self):

return "Truck"

# Polymorphism in action:

def display\_info(self):

print(f"This is a {self.get\_type()} - a {self.color} {self.make} {self.model}.")

truck1 = Truck("Ford", "F-150", "Black")

sedan2 = Sedan("Toyota", "Corolla", "White", 4)

truck1.display\_info() # Calls Truck's version: "This is a Truck - a Black Ford F-150."

sedan2.display\_info() # Calls Sedan's version: "This is a Sedan - a White Toyota Corolla."

Even though truck1 and sedan2 are different types, they can both respond to the same display\_info() method call, demonstrating polymorphism.

**What is Abstraction?**

**Abstraction** is a fundamental principle of Object-Oriented Programming (OOP) that involves simplifying complex systems by modeling classes based on a high-level view of their functionality, rather than their specific, low-level implementation details. It focuses on what an object **does**, hiding the how. The goal is to reduce complexity and allow programmers to focus on the essential features of a problem without getting bogged down in the minutiae.

Think of it like using a smartphone. You interact with icons and buttons on the screen (**the abstraction**), which allows you to make calls, send messages, and browse the internet. You don't need to know the intricate details of how the phone's internal hardware and software work together to perform these tasks. The complex inner workings are **abstracted away** from you, the user.

**Abstraction vs. Encapsulation**

While often used together, **abstraction** and **encapsulation** are distinct concepts:

* **Abstraction** focuses on **hiding complexity at the design level**. It deals with providing a simplified interface to a complex system. It's about showing only what's necessary.
* **Encapsulation** focuses on **hiding complexity at the implementation level**. It deals with bundling data and the methods that operate on that data into a single unit (a class). It's about protecting data from unauthorized access.

A good way to remember the difference is: **abstraction** is the "what" (what an object does), while **encapsulation** is the "how" (how the object protects its data).

**How is Abstraction Implemented?**

Abstraction is primarily implemented in programming through **abstract classes** and **interfaces**.

**1. Abstract Classes**

An **abstract class** is a class that cannot be instantiated (you can't create an object from it directly). It's designed to be a base class for other classes to inherit from. It can contain both concrete methods with implementation and **abstract methods**, which are declared but not defined. Subclasses must provide the implementation for all abstract methods.

**Example (Python):**

Python

from abc import ABC, abstractmethod

# The 'Vehicle' class is an abstract class

class Vehicle(ABC):

@abstractmethod

def start\_engine(self):

pass

@abstractmethod

def stop\_engine(self):

pass

def refuel(self):

print("Refueling the vehicle.")

# 'Car' is a concrete class that must implement the abstract methods

class Car(Vehicle):

def start\_engine(self):

print("Car engine started.")

def stop\_engine(self):

print("Car engine stopped.")

# This will raise an error:

# vehicle = Vehicle()

# This works:

my\_car = Car()

my\_car.start\_engine() # Output: Car engine started.

my\_car.refuel() # Output: Refueling the vehicle.

In this example, the Vehicle class defines the **concept** of a vehicle's behavior (start\_engine and stop\_engine) without providing the implementation. The Car class then provides the specific **details** of how a car's engine starts and stops.

**2. Interfaces**

An **interface** is similar to an abstract class but is even more abstract. It defines a contract—a set of method signatures that a class must implement. An interface contains no implemented methods or data fields; it's just a pure blueprint of what a class should be able to do.

**Example (Java):**

Java

// 'Shape' is an interface

public interface Shape {

double getArea();

double getPerimeter();

}

// 'Circle' is a class that implements the 'Shape' interface

public class Circle implements Shape {

private double radius;

public Circle(double radius) {

this.radius = radius;

}

@Override

public double getArea() {

return Math.PI \* radius \* radius;

}

@Override

public double getPerimeter() {

return 2 \* Math.PI \* radius;

}

}

Here, the Shape interface guarantees that any class implementing it will have getArea() and getPerimeter() methods. The specific implementation of these methods, however, is left up to the concrete classes like Circle. This allows for **polymorphism** where you can treat a Circle object as a Shape object, relying only on the abstract interface.

|  |  |  |
| --- | --- | --- |
| Concept | Meaning | Example |
| Class | Blueprint for objects | class Car: |
| Object | Instance of class | car1 = Car("BMW") |
| Attributes | Variables inside class | car1.brand |
| Methods | Functions inside class | car1.drive() |
| Encapsulation | Data hiding | self.\_\_balance |
| Inheritance | Reusing class features | class Car(Vehicle) |
| Polymorphism | Same method, different behavior | animal.sound() |
| Abstraction | Hiding implementation | abstractmethod |

A **constructor** is a special method in a class that's automatically called when you create a new object from that class. Its main job is to **initialize** the new object, setting up its initial state by assigning values to its attributes.

**How it Works in Python**

In Python, the constructor is a method with the specific name \_\_init\_\_. The \_\_init\_\_ method is often referred to as the constructor, though technically, it's an initializer. The first argument of this method is always self, which is a reference to the newly created object.

**Example in Python**

Let's use the Car class from earlier. The \_\_init\_\_ method is the constructor that gives each car object its unique make, model, and color when it's created.

Python

class Car:

# This is the constructor

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

self.make = make # Initializing the 'make' attribute

self.model = model # Initializing the 'model' attribute

self.color = color # Initializing the 'color' attribute

def start\_engine(self):

print(f"The {self.color} {self.make} {self.model}'s engine is starting.")

# When we create an object, the constructor is called automatically

car1 = Car("Toyota", "Camry", "Red")

car2 = Car("Honda", "Civic", "Blue")

# The objects are now fully initialized and ready to use

car1.start\_engine()

car2.start\_engine()

In this code, \_\_init\_\_(self, make, model, color) is the constructor. When you write car1 = Car("Toyota", "Camry", "Red"), Python automatically calls the \_\_init\_\_ method, passing "Toyota", "Camry", and "Red" as arguments, which are then used to set the attributes of the new car1 object.

**Initialization** means giving an initial or starting value to something, like a variable or an object's attributes. In the context of programming and Object-Oriented Programming (OOP), it's the process of setting up the first state of an object right after it's created.

**Initialization of Variables**

When you declare a variable, you often need to give it a starting value before you use it. This is initialization. Without an initial value, a variable might contain an unpredictable or "garbage" value.

* **Declaration:** my\_number;
* **Initialization:** my\_number = 10;

**Initialization of Objects**

In OOP, initialization happens within the **constructor** of a class. When a new object is created from a class blueprint, the constructor's job is to set the initial values for the object's properties or attributes. This ensures the object is in a valid, ready-to-use state from the moment it's created.

**Example in Python**

In Python, the \_\_init\_\_ method is the constructor where initialization takes place.

Python

class Dog:

# This is the constructor, responsible for initialization

def \_\_init\_\_(self, name, breed):

self.name = name # Initializing the 'name' attribute

self.breed = breed # Initializing the 'breed' attribute

def bark(self):

print(f"{self.name} says woof!")

# The object 'my\_dog' is created and initialized in one step

my\_dog = Dog("Buddy", "Golden Retriever")

# The attributes are now initialized and can be used

print(my\_dog.name) # Output: Buddy

my\_dog.bark() # Output: Buddy says woof!

In this example, when my\_dog = Dog("Buddy", "Golden Retriever") is executed, the \_\_init\_\_ method is called. This method takes the arguments "Buddy" and "Golden Retriever" and uses them to **initialize** the name and breed attributes of the my\_dog object.