SOCHSPACE

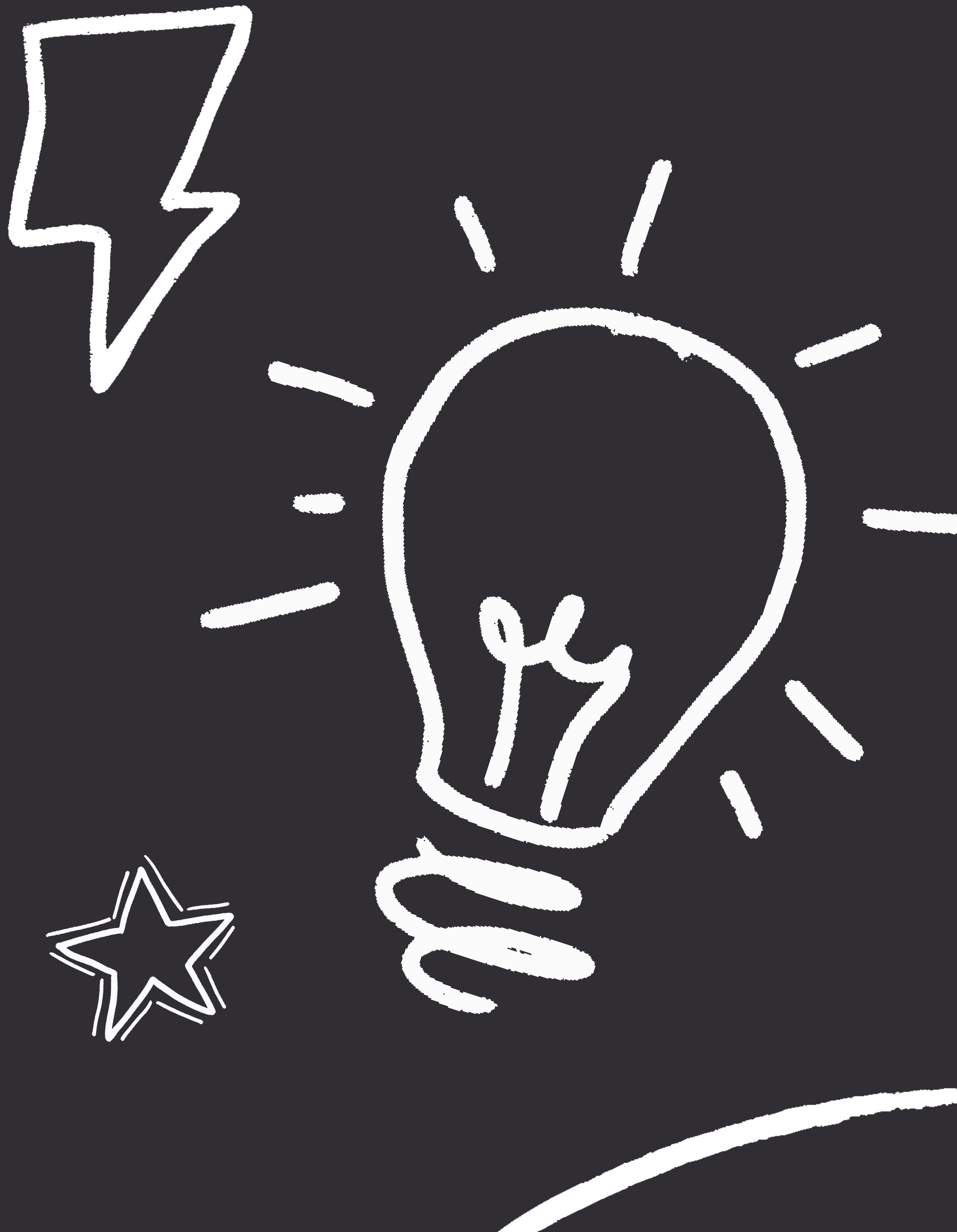
Object Oriented Programming

Presenter:

KIRAN HAYAT

Date:

5 August 2024



What is Object-Oriented Programming?

Object-Oriented Programming (OOP) is an approach to writing programs by creating classes and objects. This method focuses more on data rather than logic.

Why Object-Oriented Programming?

To solve real-world problems, Object-Oriented Programming (OOP) provides several features:

- Inheritance: Reusability
- Encapsulation: Data security
- Abstraction: Data hiding

CLASS

A class is a template, blueprint, or prototype for creating objects. Every object belongs to some class.
example:
Email class : email1 , email2, email3
A class is a collection of attributes and methods. It is also a collection of objects. Technically, a class is a user-defined data type.

ATTRIBUTES

- Variables that store data or properties of an object.
- Define the state or characteristics of an object.
- Defined within a class.
- Accessed through objects created from the class.
- Example: In a Email class, attributes might include heading, participants, attachments.

METHODS

- Functions defined within a class.
- Perform operations using the object's attributes.
- Define the behavior of the class and its objects.
- Can take parameters and return values.
- Accessed using the dot notation on an object.
- Example: In the Email class, sen(), save_as_draft() is a method.

OBJECTS

- An object represents real-life entities.
- Examples: Email, man, student, employee, etc.
- Every object has two properties:
  Attributes (variables)
  Behaviors (methods)

CREATING CLASS & OBJECTS

```
class ClassName:
        #attributes-->variables
        #methods-->functions
object1=ClassName([arg])
object2=ClassName([arg])
```

Example

```
class Email:
        pass
e1=Email()
e2=Email()
print(type(e1))
```

output

```
<class '__main__.Email'>
```

CONSTRUCTORS & TYPES OF CONSTRUCTOR

- Definition: A special method used for initializing objects with attributes.
- Method Name: __init__
- First Argument: self (refers to the instance of the class)

Parameterized Constructor:

- Takes additional parameters besides self.
- Example: __init__(self, name, age)

Non-Parameterized Constructor:

- Only takes self as a parameter.
- Example: __init__(self)

Default Constructor:

- Uses pass if no specific initialization is needed.
- Python's default behavior if no constructor is defined is to call a built-in default constructor.

Without Constructor:

without constructor : object cannot be created

If no constructor is defined, Python automatically provides a default constructor.

SELF

- self is a variable that contains the memory reference of the current object.
- Memory Allocation: When an object is created, memory is allocated for it.
- Memory Reference: The memory reference (address) of the object is returned and assigned to the object.
- Automatic Passing: self is automatically passed to methods in a class to refer to the current instance of the object.
- Initialization: The constructor (__init__) uses self to initialize instance variables at the memory reference of the object.
-

Working of OOP and self:

1. Creating an Object:
    - Memory is allocated for the object.
    - The object is assigned a memory reference.
2. Constructor Call:
    - The __init__ method is called with self pointing to the new object.
    - self allows access to the object's attributes and methods.
3. Instance Variables Initialization:
    - Using self, instance variables are initialized at the object's memory reference.
4. Method Calls:
    - When a method is called on an object, self is automatically passed to the method, referring to the calling object.

Class Members

- Class members include both attributes (variables) and methods (actions) defined within a class.

Attributes (Variables):

- Accessing Attributes: Use the syntax object_name.attribute_name.
- Example: employee1.name

Accessing Attributes:

- object_name.attribute_name

Methods (Actions):

- Accessing Methods: Use the syntax object_name.method_name().
- Example: employee1.display_info()

Accessing Methods:

- object_name.method_name()

Built-in Class Functions:

getattr(object, name[, default]):

- Purpose: Retrieves the value of an attribute named name from the object.
- Parameters:
- object: The object from which to get the attribute.
- name: The name of the attribute to retrieve.
- default: (Optional) A default value to return if the attribute does not exist.

setattr(object, name, value):

- Purpose: Sets the value of an attribute named name on the object.
- Parameters:
- object: The object on which to set the attribute.
- name: The name of the attribute to set.
- value: The value to assign to the attribute.

delattr(object, name):

- Purpose: Deletes the attribute named name from the object.
- Parameters:
- object: The object from which to delete the attribute.
- name: The name of the attribute to delete.

hasattr(object, name):

- Purpose: Checks if the object has an attribute named name.
- Parameters:
- object: The object to check.
- name: The name of the attribute to check for.
- Returns: True if the attribute exists, False otherwise.

Built-in Class Attributes

__dict__:

Contains a dictionary representation of the class’s namespace, including its attributes and methods.

__doc__:

Holds the documentation string (docstring) of the class. If no docstring is provided, it is None.

__name__:

Returns the name of the class as a string.

__bases__:

A tuple containing the base classes (parent classes) of the class. Useful for understanding class inheritance.

__module__:

Provides the name of the module in which the class was defined.

isinstance()

- The isinstance() function checks if an object belongs to a specific class.
- It returns True or False.
- example:

1. isinstance(e1, Employee) # True
2. isinstance(student2, Employee) # False

Usage:

```
if isinstance(object, classname):
    pass
```

Instance Variables

Types of Variables

- Instance Variables
- Class Variables

Instance Variables

Variables that are specific to each object of a class.

Example:

```
class Student:
    def __init__(self, nm, m):
        self.name = nm
        self.marks = m

s1 = Student('Kiran', 34)
```

- name and marks are instance variables.
- Their values are unique to each instance of the Student class.

Characteristics:

1. Created for each instance: Each object has its own separate instance variables.
2. Separate Copy: Every object has its own copy of the instance variables.
3. Different Values: The values of instance variables differ from one object to another.
4. No Cross-Effect: Modifying an instance variable in one object does not affect other objects.

Class Variables

- Variables that are shared among all instances of a class.
- Characteristics:

1. Shared Across Class: Made for the entire class and shared among all objects.
2. Single Copy: Only one copy of the class variable is created and distributed to all objects.
3. Modification Impact: Modifying a class variable affects all instances of the class.

Example:

```
class Employee:
    company_name = 'TCS'  # Class variable


# Modifying a class variable
Employee.company_name = 'Infosys'
```

Class Methods

Methods that operate on class variables and are bound to the class rather than instances.

Characteristics:

1. First Argument: The first argument is a reference to the class itself, typically named cls.
2. Decorator: Defined using the @classmethod decorator.

Example:

```
class Employee:
    company_name = 'TCS'  # Class variable
    @classmethod
    def change_company(cls, new_name):
        cls.company_name = new_name

# Modifying a class variable using a class method
Employee.change_company('Infosys')
```

Instance Methods: Setters and Getters

- Methods that operate on instance variables. They are called on instances of a class.

1. Types of Instance Methods:
2. Setter Method: Sets the values of instance variables.
3. Getter Method: Retrieves the values of instance variables.

Example

```
class Employee:
    # Setter method
    def setName(self, name):
        self.name = name

    # Getter method
    def getName(self):
        print("The name is:", self.name)

# Creating an instance of Employee
e1 = Employee()

# Setting the name using the setter method
e1.setName("Kiran")

# Getting the name using the getter method
e1.getName()
```

output

```
The name is: Kiran
```

Static Method

- External Data Operations: Static methods perform operations on external data, meaning data that does not belong to any specific class or object.
- Class Data Access: They can also perform operations on class data (also known as static data).
- No Object or Class Reference Needed: Static methods do not need to pass a reference to an object (self) or class (cls).
- Created with @staticmethod Decorator: Static methods are defined using the @staticmethod decorator.

Example: Static Method in a Class

```
class Bank:
    bank_name = 'HBL'
    rate_of_interest = 12.25

    @staticmethod
    def simple_interest(principle, n):
        si = (principle * n * Bank.rate_of_interest) / 100
        return si

# Calling the static method without creating an instance of the class
interest = Bank.simple_interest(2000, 3)
print("Simple Interest is:", interest)
```

Output:

```
Simple Interest is: 735.0
```

Inheritance in Python

- Definition: Inheritance is a mechanism in Python that allows a new class (child class) to inherit attributes and methods from an existing class (parent class).
- Terminology:

1. Parent Class (Base Class, Existing Class, Superclass): The class whose attributes and methods are inherited by another class.
2. Child Class (Subclass, Derived Class): The class that inherits from the parent class and can have additional attributes and methods.

- Object Class: All classes in Python are derived from the built-in object class, which is the base class for all new-style classes.
- Creating a Child Class: To create a child class, you define it by specifying the parent class in parentheses.

Example

```
# Parent class
class Parent:
    def __init__(self, name):
        self.name = name

    def greet(self):
        print(f"Hello, I am {self.name}.")

# Child class inheriting from Parent class
class Child(Parent):
    def __init__(self, name, age):
        super().__init__(name)  # Initialize attributes from the Parent class
        self.age = age

    def show_age(self):
        print(f"I am {self.age} years old.")

# Creating an instance of the Child class
child_instance = Child("Kiran", 25)

# Calling methods from both Parent and Child classes
child_instance.greet()      # Method from Parent class
child_instance.show_age()   # Method from Child class
```

Need for Inheritance

- Code Reusability:

Write Once, Use Many Times: Inheritance allows you to write code once in a parent class and reuse it in multiple child classes. This reduces redundancy and promotes a more organized and manageable codebase.

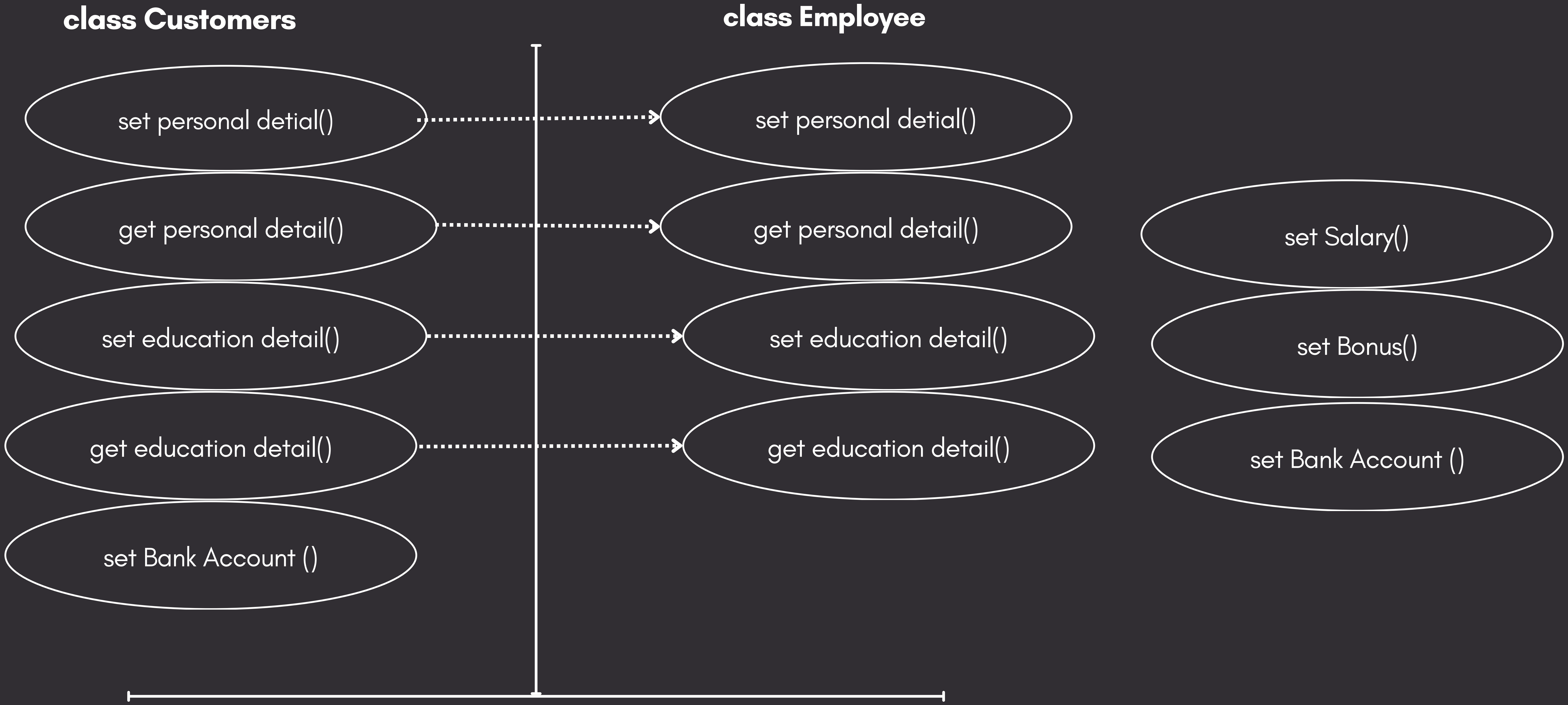
- Hierarchical Relationships:

Class Relationships: When classes have a hierarchical relationship, inheritance helps model these relationships effectively. For instance, a base class can represent a general concept, while derived classes represent more specific instances of that concept.

- Extensibility:

Easy Extension: Inheritance allows new functionality to be added to existing classes without modifying the original class. This makes it easier to extend and maintain code.

Example: Bank Management System



Constructor in Inheritance

- Concept: In inheritance, the constructor (__init__ method) of a parent class is not automatically called by the child class. You need to explicitly call the parent class's constructor if you want its initialization logic to be executed.
- How It Works:
- By Default: The constructor of the parent class is not automatically invoked in the child class.
- Explicit Call: To ensure the parent class's constructor is executed, you need to explicitly call it using super().

super() Function

- Purpose: The super() function is used to access methods and attributes from a parent (or superclass) in a child (or subclass) class. It returns a temporary object that represents the parent class and allows you to call methods or access attributes from the parent class.
- Benefits:

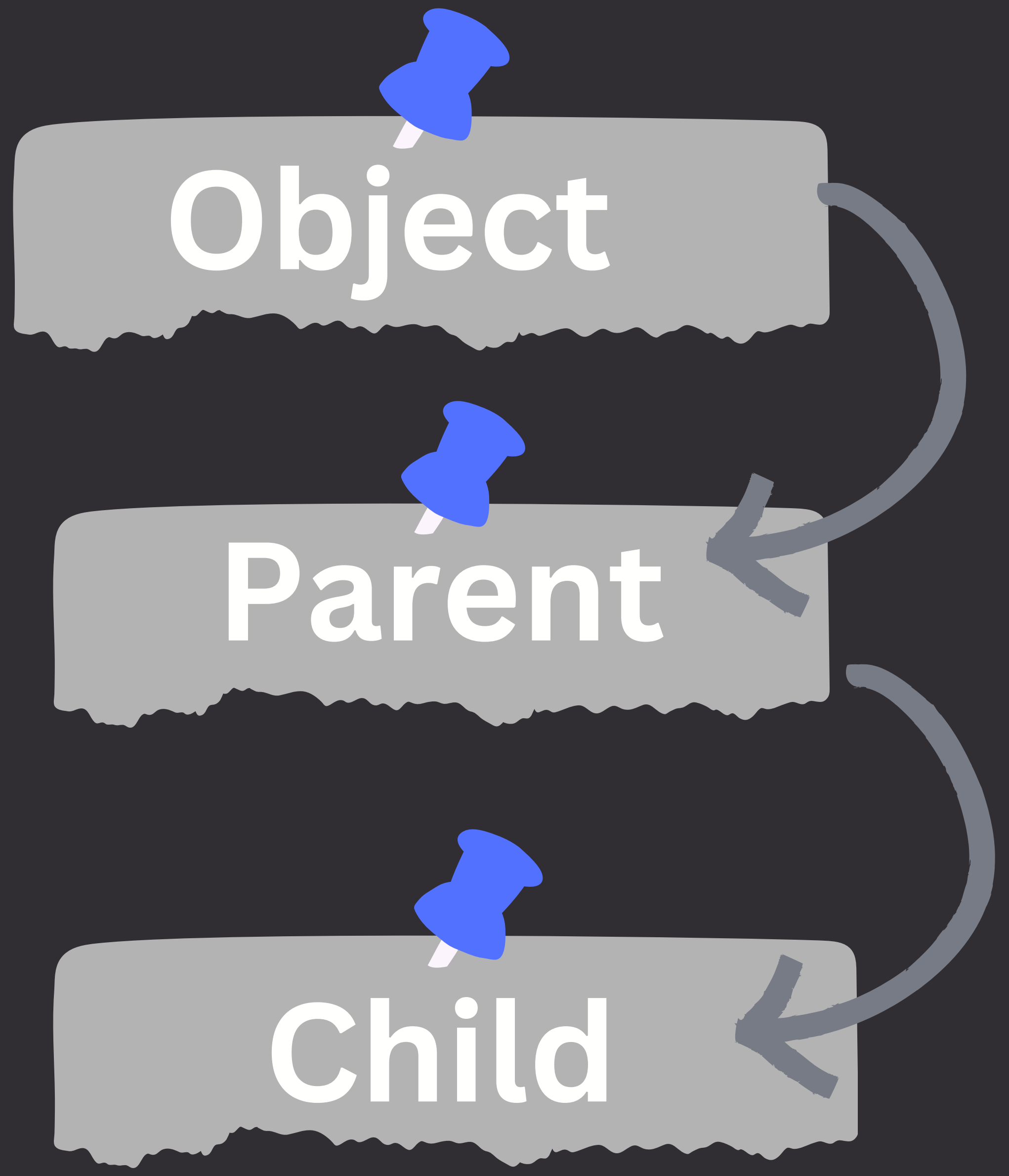
1. Manageability: Simplifies accessing parent class methods and attributes, making inheritance more manageable.
2. Maintainability: Helps ensure that the parent class is properly initialized and methods are correctly overridden.

Types of Inheritance

- Single Inheritance
- Multi-Level Inheritance
- Hierarchical Inheritance
- Hybrid Inheritance
- Cyclic Inheritance

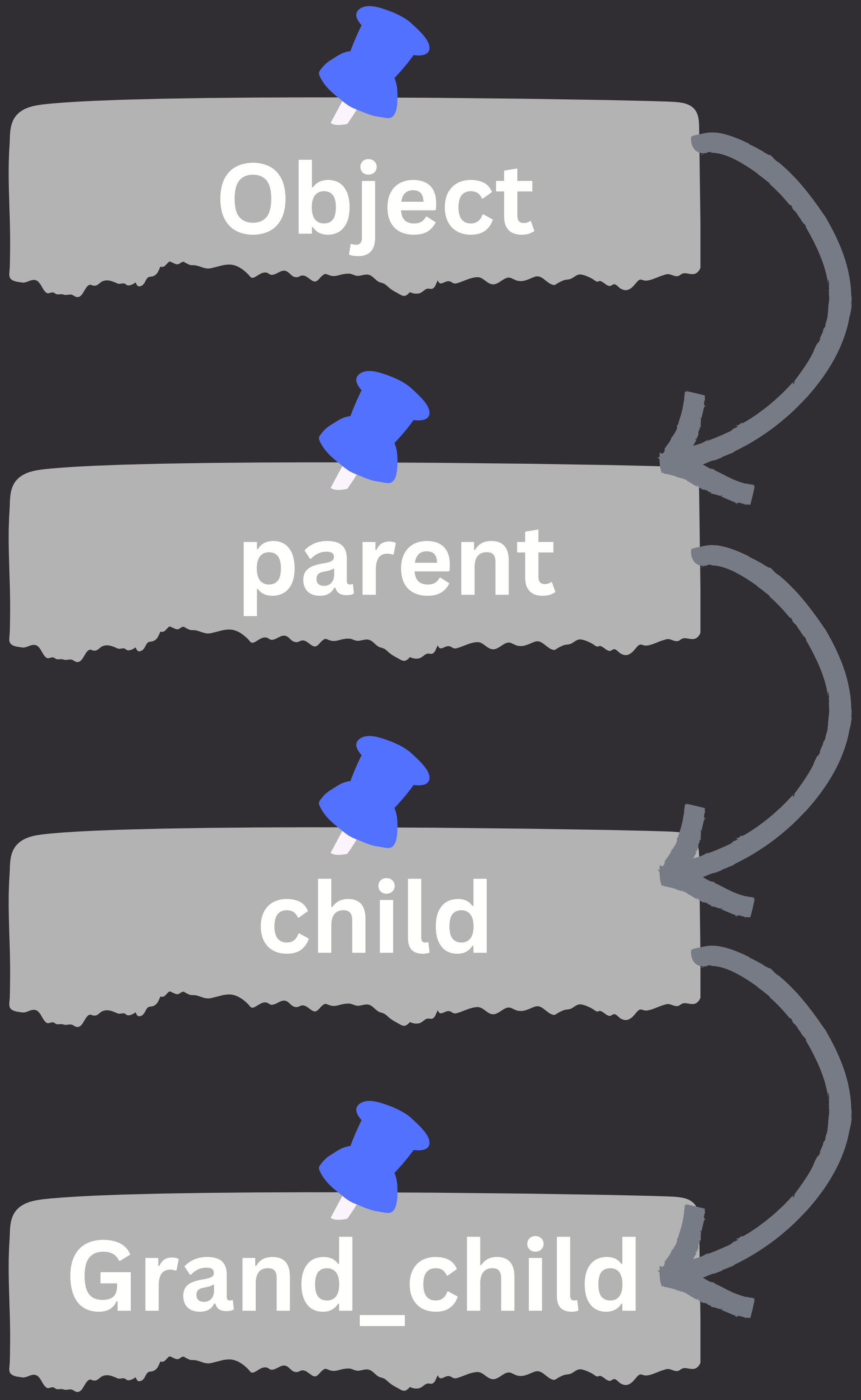
Single Inheritance

In single inheritance, a child class inherits from a single parent class.



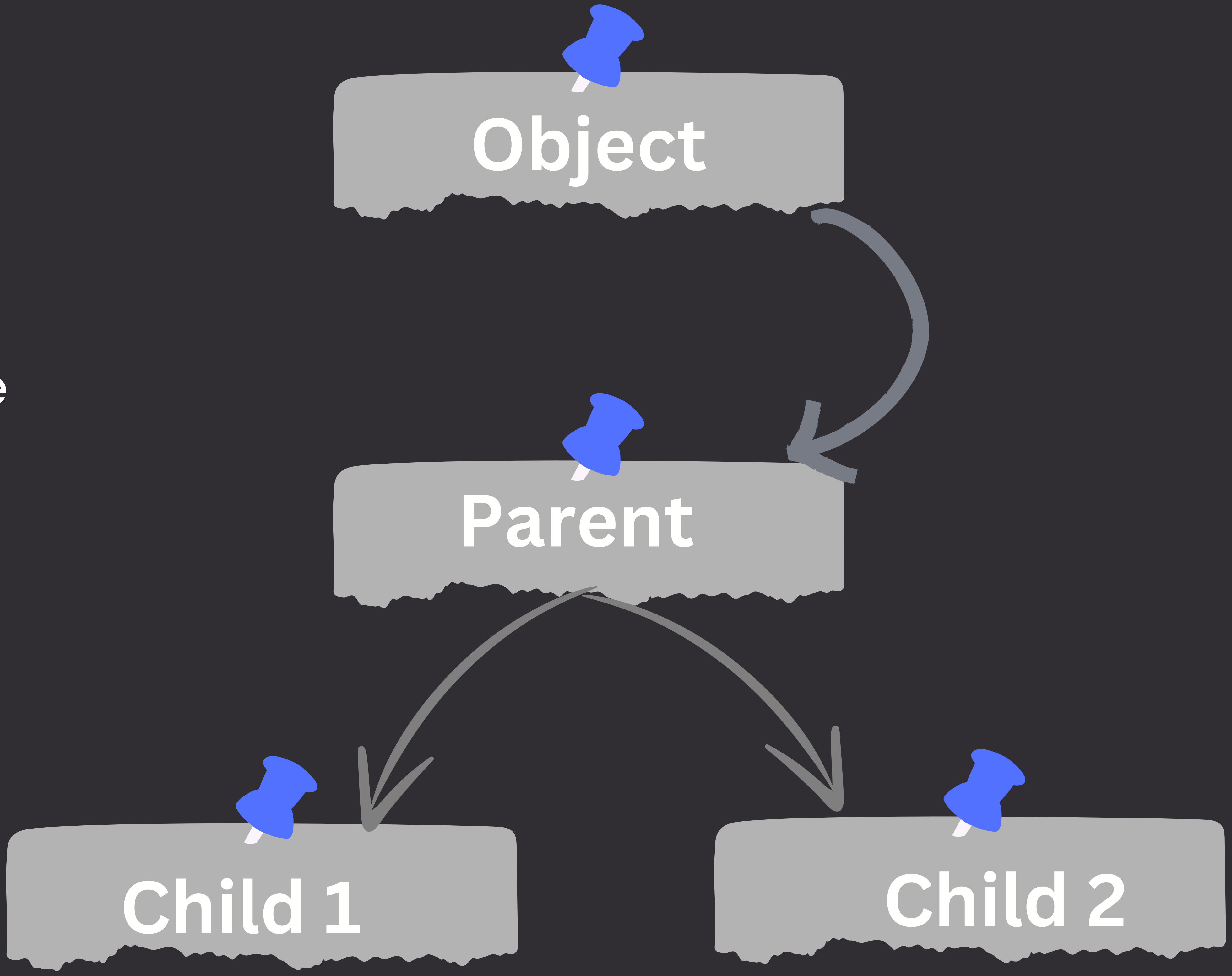
Multi-Level Inheritance:

In multi-level inheritance, a child class inherits from a parent class, and another class inherits from that child class, creating a chain of inheritance.



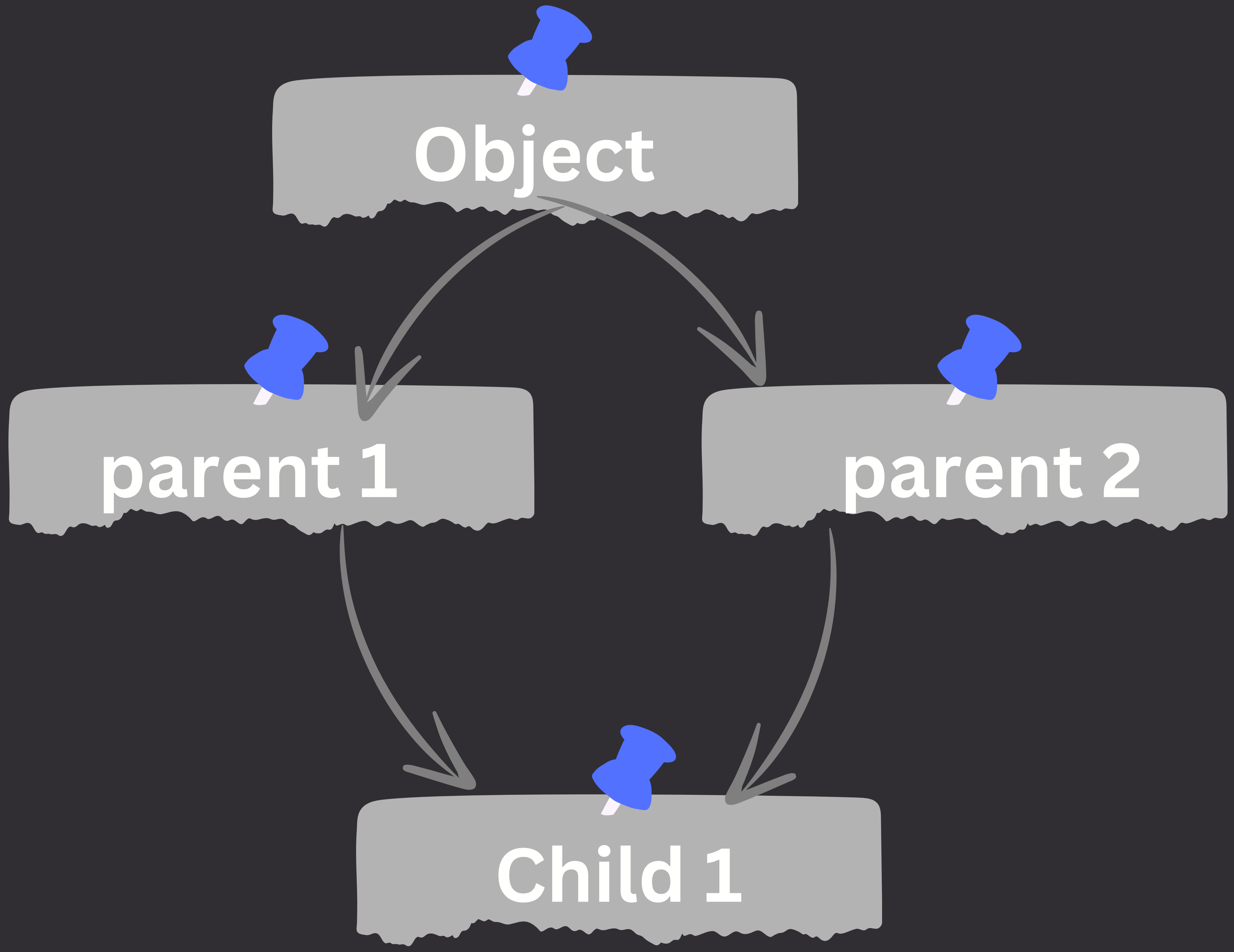
Hierarchical Inheritance:

In hierarchical inheritance, multiple child classes inherit from a single parent class.



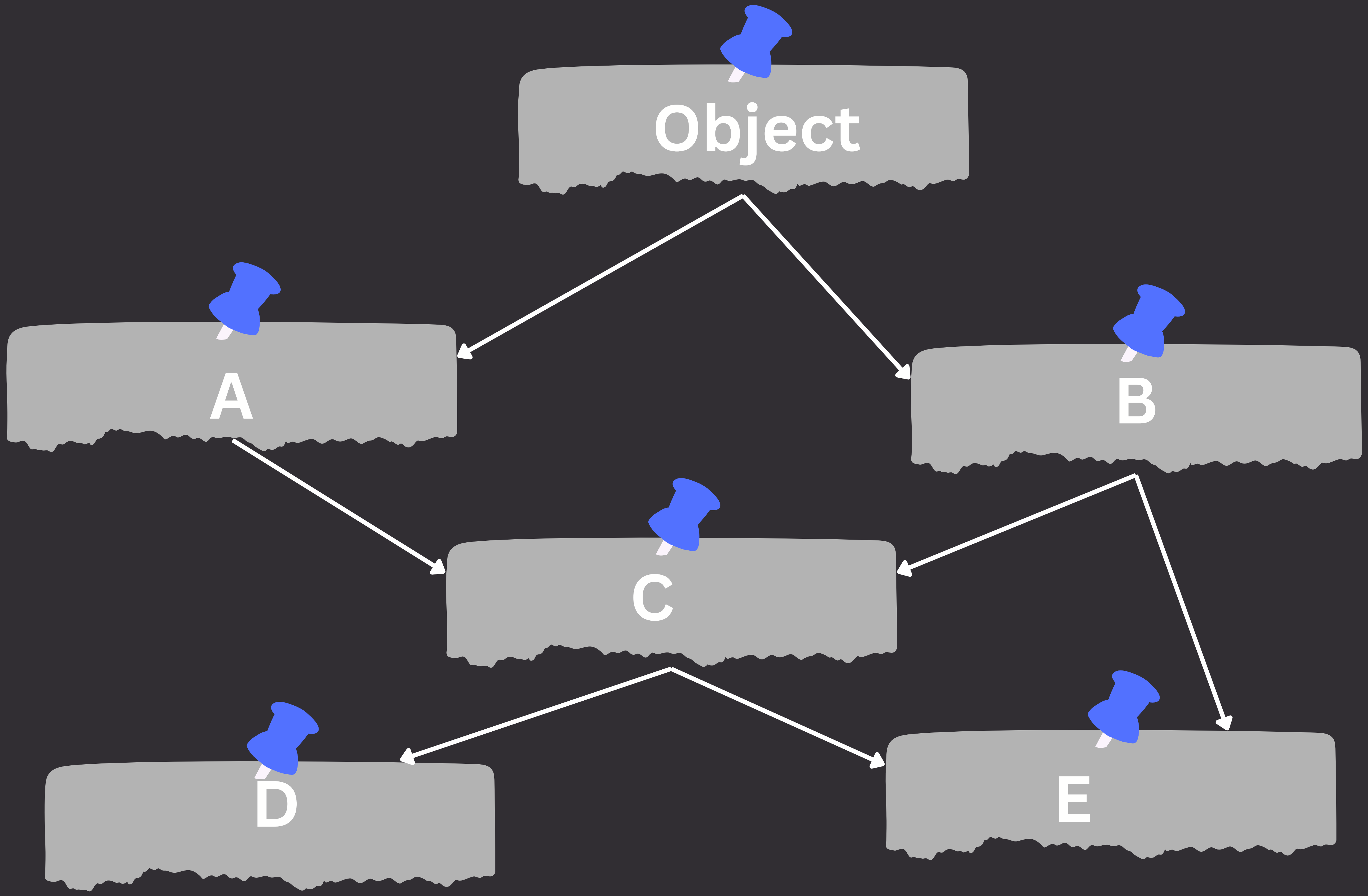
Multiple Inheritance:

In Multiple inheritance, a child class inherit from multiple parent classes.



hybrid Inheritance:

In hybrid inheritance, it contains multiple types of inheritance



Thank You!

Thank you for exploring Object-Oriented Programming with me. I hope this presentation has sparked your interest and inspired your coding journey. Happy coding!