Object Oriented Programming

Key Principles of OOP

- Encapsulation: Grouping related variables and functions that operate on them into objects.
- Inheritance: Creating new classes from existing ones, promoting code reuse.
- Polymorphism: Writing code that can work with objects of multiple types.
- Abstraction: Hiding implementation details and showing only the necessary features.

Study Tips

- Practice writing classes and creating objects.
- Implement simple projects using OOP principles.
- Understand how inheritance, polymorphism, encapsulation, and abstraction work in practice.
- Review sample interview questions and solutions related to OOP in Python.

Fundamentals

Classes and Objects

Class: A blueprint for creating objects (a particular data structure), defining attributes and methods.

```
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def bark(self):
        return f"{self.name} says woof!"
```

Object: Instance of a class

```
my_dog = Dog("Buddy", 3)
print(my_dog.bark())  # Output: Buddy says woof!
```

Attributes and Methods

Attributes: Variables that belong to a class (class attributes) or an instance of a class (instance attributes).

```
class Car:
    wheels = 4  # Class attribute

    def __init__(self, make, model):
```

```
        self.make = make  # Instance attribute
        self.model = model
```

Methods: Functions defined within a class that describe the behaviors of the objects of the class.

```
class Car:
    def start(self):
        return "The car is starting"
```

The `__init__` Method

Constructor: The **init** method is called when an object is instantiated, allowing you to initialize attributes.

```
class Student:
    def __init__(self, name, grade):
        self.name = name
        self.grade = grade
```

Inheritance

Allows a class (child) to inherit attributes and methods from another class (parent).

```
class Animal:
    def __init__(self, species):
        self.species = species

    def make_sound(self):
        pass

class Dog(Animal):
    def __init__(self, name, age):
        super().__init__('Dog')
        self.name = name
        self.age = age

    def make_sound(self):
        return "Woof!"

my_dog = Dog("Buddy", 3)
print(my_dog.make_sound())  # Output: Woof!
```

Encapsulation

Encapsulation restricts access to certain components of an object and prevents the accidental modification of data. Use underscores to indicate private attributes.

```
class BankAccount:
    def __init__(self, owner, balance):
        self.owner = owner
        self.__balance = balance  # Private attribute

    def deposit(self, amount):
        self.__balance += amount

    def withdraw(self, amount):
        if amount <= self.__balance:
            self.__balance -= amount
            return amount
        return "Insufficient funds"

    def get_balance(self):
        return self.__balance
```

Polymorphism

Allows objects of different classes to be treated as objects of a common superclass. It provides a way to use a common interface for multiple forms (data types).

```
class Cat(Animal):
    def make_sound(self):
        return "Meow!"

animals = [Dog("Buddy", 3), Cat("Whiskers")]

for animal in animals:
    print(animal.make_sound())  # Output: Woof! Meow!
```

Abstraction

Abstraction involves hiding the complex implementation details and showing only the necessary features of an object. This can be achieved using abstract classes (from the `abc` module)

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height
```

```
    def area(self):
        return self.width * self.height

rect = Rectangle(10, 20)
print(rect.area())  # Output: 200
```

Abstraction Specifics

An abstract class in object-oriented programming (OOP) serves as a blueprint for other classes. It allows you to define methods that must be created within any child classes built from the abstract class. Abstract classes cannot be instantiated on their own and are used to provide a common interface for all the subclasses.

Key Points of Abstract Classes: Enforcing a Contract: Abstract classes define a set of methods that must be implemented by any subclass. This ensures that certain methods are always present in the subclasses, providing a consistent interface.

Partial Implementation: Abstract classes can contain both concrete methods (with implementation) and abstract methods (without implementation). This allows for a shared base functionality while leaving specific details to be implemented by subclasses.

Promoting Code Reusability: By defining common behavior in an abstract class, you can avoid code duplication in subclasses. Subclasses inherit the common behavior and only implement the specific behavior.

Example in Python In Python, you can define an abstract class using the abc module. Here is an example:

```
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def sound(self):
        pass

    def move(self):
        print("Moving...")

class Dog(Animal):
    def sound(self):
        return "Woof!"

class Cat(Animal):
    def sound(self):
        return "Meow!"

# This will raise an error
# animal = Animal()

dog = Dog()
print(dog.sound())  # Output: Woof!
```

```
dog.move()          # Output: Moving...

cat = Cat()
print(cat.sound())  # Output: Meow!
cat.move()          # Output: Moving...
```

Explanation:

1. Abstract Class Definition:

Animal is an abstract class that inherits from ABC (Abstract Base Class). The sound method is an abstract method, meaning it has no implementation in the Animal class and must be implemented in any subclass. The move method is a concrete method with an implementation that will be inherited by subclasses.

2. Concrete Subclasses:

Dog and Cat are concrete subclasses that inherit from Animal. Both Dog and Cat provide implementations for the sound method.

3. Instantiation:

Trying to instantiate Animal directly will raise an error because it contains abstract methods. Dog and Cat can be instantiated because they provide implementations for all abstract methods defined in Animal.

Benefits of Using Abstract Classes:

- Enforced Consistency: Ensures that all subclasses follow a certain structure, making it easier to understand and maintain the code.
- Code Reusability: Common methods and properties can be defined once in the abstract class and reused across multiple subclasses.
- Clear Design: Abstract classes help to define a clear and consistent interface for a family of related classes, which can improve the design and readability of the code.

Abstract classes are a powerful tool in OOP that help to create a well-defined and consistent code structure, promoting good design principles and code reuse.

Dunder Methods

Methods like .**init**() and .**str**() are called dunder methods because they begin and end with double underscores. There are many dunder methods that you can use to customize classes in Python. Understanding dunder methods is an important part of mastering object-oriented programming in Python, but for your first exploration of the topic, you'll stick with these two dunder methods.

`__str__():`

```
# dog.py

class Dog:
    # ...
```

```
    def __str__(self):
        return f"{self.name} is {self.age} years old"

>>> miles = Dog("Miles", 4)
>>> print(miles)
'Miles is 4 years old'
```

`__init__()`:

```
# dog.py

class Dog:
    species = "Canis familiaris"

    def __init__(self, name, age):
        self.name = name
        self.age = age

>>> miles = Dog("Miles", 4)
>>> buddy = Dog("Buddy", 9)
```

`__repr__()`:

The reason there are two methods to display an object is that they have different purposes:

- .**repr**() provides the official string representation of an object, aimed at the programmer.
- .**str**() provides the informal string representation of an object, aimed at the user. The target audience for the string representation returned by .**repr**() is the programmer developing and maintaining the program. In general, it provides detailed and unambiguous information about the object. Another important property of the official string representation is that a programmer can normally use it to re-create an object equal to the original one.

The .**str**() method provides a string representation targeted to the program's user, who may not necessarily be a Python programmer. Therefore, this representation enables any user to understand the data contained in the object. Usually, it's simpler and easier to read for a user.

```
# book.py

class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author

    def __repr__(self):
        class_name = type(self).__name__
        return f"{class_name}(title={self.title!r}, author=
{self.author!r})"
```

```
    def __str__(self):
        return f'"{self.title}" by {self.author}'

odyssey = Book("The Odyssey", "Homer")

print(repr(odyssey))
print(str(odyssey))
```

`__iter__():`

```
class ThreeDPoint:
    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z
    def __iter__(self):
        yield from (self.x, self.y, self.z)


>>> list(ThreeDPoint(4, 8, 16))
[4, 8, 16]
```

This class takes three arguments representing the space coordinates of a given point. The .**iter**() method is a generator function that returns an iterator. The resulting iterator yields the coordinates of ThreeDPoint on demand.

The call to list() iterates over the attributes .x, .y, and .z, returning a list object. You don't need to call .**iter**() directly. Python calls it automatically when you use an instance of ThreeDPoint in an iteration.

`__eq__:`

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __eq__(self, other):
        return self.name == other.name and self.age == other.age

x = Person('Mike', 25)
y = Person('Sarah', 27)
z = Person('Mike', 25)

print(x==z)
```

`@classmethod`

```python
# point.py

class ThreeDPoint:
    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z

    def __iter__(self):
        yield from (self.x, self.y, self.z)

    @classmethod
    def from_sequence(cls, sequence):
        return cls(*sequence)

    def __repr__(self):
        return f"{type(self).__name__}({self.x}, {self.y}, {self.z})"
```

Another example of how to use `@classmethod`

```python
class Deck:
    def __init__(self, cards):
        self.cards = cards

    @classmethod
    def create(cls, shuffle=False):
        """Create a new deck of 52 cards"""
        cards = [Card(s, r) for r in Card.RANKS for s in Card.SUITS]
        if shuffle:
            random.shuffle(cards)
        return cls(cards)

    def deal(self, num_hands):
        """Deal the cards in the deck into a number of hands"""
        cls = self.__class__
        return tuple(cls(self.cards[i::num_hands]) for i in
range(num_hands))

# ...

class Game:
    def __init__(self, *names):
        """Set up the deck and deal cards to 4 players"""
        deck = Deck.create(shuffle=True)
        self.names = (list(names) + "P1 P2 P3 P4".split())[:4]
        self.hands = {
            n: Player(n, h) for n, h in zip(self.names, deck.deal(4))
        }
```

In the .from_sequence() class method, you take a sequence of coordinates as an argument, create a ThreeDPoint object from it, and return the object to the caller. To create the new object, you use the cls argument, which holds an implicit reference to the current class, which Python injects into your method automatically.

`@staticmethod`

Your Python classes can also have static methods. These methods don't take the instance or the class as an argument. So, they're regular functions defined within a class. You could've also defined them outside the class as stand-alone function.

You'll typically define a static method instead of a regular function outside the class when that function is closely related to your class, and you want to bundle it together for convenience or for consistency with your code's API. Remember that calling a function is a bit different from calling a method. To call a method, you need to specify a class or object that provides that method.

```
# point.py

class ThreeDPoint:
    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z

    def __iter__(self):
        yield from (self.x, self.y, self.z)

    @classmethod
    def from_sequence(cls, sequence):
        return cls(*sequence)

    @staticmethod
    def show_intro_message(name):
        print(f"Hey {name}! This is your 3D Point!")

    def __repr__(self):
        return f"{type(self).__name__}({self.x}, {self.y}, {self.z})"
```

Practice Problem: Vehicle Inventory System

You are tasked with creating a system to manage a car dealership's inventory. The system should be able to:

1. Add new vehicles to the inventory.
2. Remove vehicles from the inventory by their VIN (Vehicle Identification Number).
3. Retrieve information about a specific vehicle by its VIN.
4. List all vehicles in the inventory, with the ability to filter by type (e.g., car, truck, motorcycle).

Requirements

Vehicle Class

Create a base class `Vehicle` with the following attributes:

- `vin`: A unique identifier for the vehicle.
- `make`: The manufacturer of the vehicle.
- `model`: The model of the vehicle.
- `year`: The year the vehicle was manufactured.
- `mileage`: The mileage of the vehicle.
- `price`: The price of the vehicle.

The `Vehicle` class should have a method `get_info` that returns a string containing the vehicle's details.

Car Class

Create a subclass `Car` that inherits from `Vehicle` and adds the following attributes:

- `doors`: The number of doors on the car.
- `convertible`: A boolean indicating whether the car is a convertible.

Truck Class

Create a subclass `Truck` that inherits from `Vehicle` and adds the following attributes:

- `bed_length`: The length of the truck's bed.
- `four_wheel_drive`: A boolean indicating whether the truck has four-wheel drive.

Motorcycle Class

Create a subclass `Motorcycle` that inherits from `Vehicle` and adds the following attribute:

- `engine_cc`: The engine displacement in cubic centimeters.

Inventory Class

Create a class `Inventory` to manage the vehicle inventory with the following methods:

- `add_vehicle(vehicle)`: Adds a vehicle to the inventory.
- `remove_vehicle(vin)`: Removes a vehicle from the inventory by its VIN.
- `get_vehicle_info(vin)`: Retrieves information about a specific vehicle by its VIN.
- `list_vehicles(vehicle_type=None)`: Lists all vehicles in the inventory. If `vehicle_type` is specified, it should only list vehicles of that type (e.g., 'Car', 'Truck', 'Motorcycle').

Example Usage

Here is an example of how the classes should be used:

```
class Vehicle:
    def __init__(self, vin, make, model, year, mileage, price):
```

```
        self.vin = vin
        self.make = make
        self.model = model
        self.year = year
        self.mileage = mileage
        self.price = price

    def get_info(self):
        return f"VIN: {self.vin}, Make: {self.make}, Model: {self.model},
Year: {self.year}, Mileage: {self.mileage}, Price: ${self.price:.2f}"

class Car(Vehicle):
    def __init__(self, vin, make, model, year, mileage, price, doors,
convertible):
        super().__init__(vin, make, model, year, mileage, price)
        self.doors = doors
        self.convertible = convertible

    def get_info(self):
        info = super().get_info()
        return f"{info}, Doors: {self.doors}, Convertible:
{self.convertible}"

class Truck(Vehicle):
    def __init__(self, vin, make, model, year, mileage, price, bed_length,
four_wheel_drive):
        super().__init__(vin, make, model, year, mileage, price)
        self.bed_length = bed_length
        self.four_wheel_drive = four_wheel_drive

    def get_info(self):
        info = super().get_info()
        return f"{info}, Bed Length: {self.bed_length} feet, Four Wheel
Drive: {self.four_wheel_drive}"

class Motorcycle(Vehicle):
    def __init__(self, vin, make, model, year, mileage, price, engine_cc):
        super().__init__(vin, make, model, year, mileage, price)
        self.engine_cc = engine_cc

    def get_info(self):
        info = super().get_info()
        return f"{info}, Engine CC: {self.engine_cc}"

class Inventory:
    def __init__(self):
        self.vehicles = []

    def add_vehicle(self, vehicle):
        self.vehicles.append(vehicle)

    def remove_vehicle(self, vin):
        self.vehicles = [v for v in self.vehicles if v.vin != vin]
```

```
    def get_vehicle_info(self, vin):
        for vehicle in self.vehicles:
            if vehicle.vin == vin:
                return vehicle.get_info()
        return "Vehicle not found"

    def list_vehicles(self, vehicle_type=None):
        if vehicle_type:
            return [v.get_info() for v in self.vehicles if
v.__class__.__name__ == vehicle_type]
        return [v.get_info() for v in self.vehicles]

# Example usage
inventory = Inventory()

# Adding vehicles
car1 = Car('1HGCM82633A004352', 'Honda', 'Accord', 2020, 15000, 20000, 4,
False)
truck1 = Truck('1FTFW1E57JFA30456', 'Ford', 'F-150', 2018, 30000, 25000,
6, True)
motorcycle1 = Motorcycle('2C3CDXBG3DH511614', 'Harley-Davidson', 'Street
750', 2019, 5000, 7500, 750)

inventory.add_vehicle(car1)
inventory.add_vehicle(truck1)
inventory.add_vehicle(motorcycle1)

# Listing all vehicles
print("All vehicles:")
for info in inventory.list_vehicles():
    print(info)

# Listing only cars
print("\nCars only:")
for info in inventory.list_vehicles('Car'):
    print(info)

# Getting information about a specific vehicle
print("\nVehicle info for VIN 1FTFW1E57JFA30456:")
print(inventory.get_vehicle_info('1FTFW1E57JFA30456'))

# Removing a vehicle
inventory.remove_vehicle('1HGCM82633A004352')
print("\nAll vehicles after removing VIN 1HGCM82633A004352:")
for info in inventory.list_vehicles():
    print(info)
```

Practice Problem: Autonomous Vehicle Fleet Management System

You are tasked with designing a fleet management system for an autonomous vehicle company. The system should be able to:

1. **Add new vehicles to the fleet.**
2. **Assign tasks to vehicles.**
3. **Track the status of each vehicle and its tasks.**
4. **Generate reports on the fleet's performance.

Requirements

Vehicle Class

Create a base class `Vehicle` with the following attributes:

- `vin`: A unique identifier for the vehicle.
- `make`: The manufacturer of the vehicle.
- `model`: The model of the vehicle.
- `year`: The year the vehicle was manufactured.
- `status`: The current status of the vehicle (e.g., "idle", "in transit", "maintenance").
- `tasks`: A list of tasks assigned to the vehicle.

The `Vehicle` class should have methods to:

- `assign_task(task)`: Assign a new task to the vehicle.
- `complete_task()`: Mark the current task as completed and update the status.
- `get_info()`: Return a string containing the vehicle's details.

Task Class

Create a `Task` class with the following attributes:

- `task_id`: A unique identifier for the task.
- `description`: A description of the task.
- `origin`: The starting location of the task.
- `destination`: The ending location of the task.
- `status`: The current status of the task (e.g., "pending", "in progress", "completed").

Fleet Class

Create a `Fleet` class to manage the fleet of vehicles with the following methods:

- `add_vehicle(vehicle)`: Add a new vehicle to the fleet.
- `remove_vehicle(vin)`: Remove a vehicle from the fleet by its VIN.
- `assign_task_to_vehicle(task, vin)`: Assign a task to a specific vehicle by its VIN.
- `complete_vehicle_task(vin)`: Mark the current task of a specific vehicle as completed.
- `get_fleet_status()`: Generate a report on the status of all vehicles and their tasks.

Example Usage

```
class Task:
    def __init__(self, task_id, description, origin, destination):
        self.task_id = task_id
        self.description = description
        self.origin = origin
        self.destination = destination
        self.status = "pending"

    def __str__(self):
        return f"Task ID: {self.task_id}, Description: {self.description},
Origin: {self.origin}, Destination: {self.destination}, Status:
{self.status}"

class Vehicle:
    def __init__(self, vin, make, model, year):
        self.vin = vin
        self.make = make
        self.model = model
        self.year = year
        self.status = "idle"
        self.tasks = []

    def assign_task(self, task):
        self.tasks.append(task)
        self.status = "in transit"
        task.status = "in progress"

    def complete_task(self):
        if self.tasks:
            current_task = self.tasks.pop(0)
            current_task.status = "completed"
            self.status = "idle" if not self.tasks else "in transit"

    def get_info(self):
        return f"VIN: {self.vin}, Make: {self.make}, Model: {self.model},
Year: {self.year}, Status: {self.status}, Tasks: {[str(task) for task in
self.tasks]}"

class Fleet:
    def __init__(self):
        self.vehicles = {}

    def add_vehicle(self, vehicle):
        self.vehicles[vehicle.vin] = vehicle

    def remove_vehicle(self, vin):
        if vin in self.vehicles:
            del self.vehicles[vin]

    def assign_task_to_vehicle(self, task, vin):
        if vin in self.vehicles:
            self.vehicles[vin].assign_task(task)
```

```
    def complete_vehicle_task(self, vin):
        if vin in self.vehicles:
            self.vehicles[vin].complete_task()

    def get_fleet_status(self):
        report = []
        for vin, vehicle in self.vehicles.items():
            report.append(vehicle.get_info())
        return "\n".join(report)

# Example usage
fleet = Fleet()

# Adding vehicles
vehicle1 = Vehicle('1HGCM82633A004352', 'Tesla', 'Model S', 2022)
vehicle2 = Vehicle('1FTFW1E57JFA30456', 'Waymo', 'Chrysler Pacifica',
2021)

fleet.add_vehicle(vehicle1)
fleet.add_vehicle(vehicle2)

# Creating tasks
task1 = Task(1, "Deliver package", "Location A", "Location B")
task2 = Task(2, "Pick up passenger", "Location C", "Location D")

# Assigning tasks to vehicles
fleet.assign_task_to_vehicle(task1, '1HGCM82633A004352')
fleet.assign_task_to_vehicle(task2, '1FTFW1E57JFA30456')

# Completing a task
fleet.complete_vehicle_task('1HGCM82633A004352')

# Generating fleet status report
print(fleet.get_fleet_status())
```

Practice Problem: Airline Ticketing System Design

Design an airline ticketing system using object-oriented principles. The system should handle flight scheduling, booking, ticket pricing, and passenger management.

Requirements

Classes

- `Airline`
- `Flight`
- `Ticket`
- `Passenger`
- `Booking`

Features

- Schedule flights with departure and arrival times, and capacity.
- Book tickets for flights, ensuring no overbooking.
- Calculate ticket prices based on factors like seat type and booking time.
- Manage passenger information and bookings.

Considerations

- Encapsulation and abstraction for handling complex operations.
- Use inheritance for different types of flights (e.g., domestic, international).
- Implement polymorphism for ticket pricing strategies (e.g., economy, business).

Implementation

Implement this system in Python using classes and appropriate methods. This exercise will help you solidify your understanding of OOP concepts and prepare for similar interview questions.

Trivia Questions

- What is the difference between inheritance and composition in object-oriented programming, and when should you prefer one over the other?
    - Answer: Inheritance and composition are two fundamental concepts in object-oriented programming that are used to build relationships between classes.
    - Inheritance is a mechanism where a new class (called a subclass) derives from an existing class (called a superclass). The subclass inherits the attributes and methods of the superclass, allowing for code reuse and the creation of a hierarchical relationship between classes. Inheritance is often described as an "is-a" relationship. For example, a `Car` class might inherit from a `Vehicle` class because a car is a type of vehicle.
    - Composition is a mechanism where a class is composed of one or more objects from other classes. This allows for a "has-a" relationship, where a class contains instances of other classes as attributes. Composition is preferred when you want to build complex types by combining objects and when you want to avoid the rigidity of inheritance hierarchies. For example, a `Car` class might have an `Engine` object because a car has an engine.
    - When to prefer one over the other:
        - Use inheritance when there is a clear hierarchical relationship and when the subclass can naturally extend the behavior of the superclass. Use composition when you want to create complex objects from simpler ones and when you want to design flexible systems where components can be easily replaced or changed.

- What are abstract classes and interfaces in object-oriented programming, and how do they differ?
    - An abstract class is a class that cannot be instantiated on its own and is meant to be subclassed. It can include both abstract methods (methods without implementation) and concrete methods (methods with implementation). Abstract methods defined in an abstract class must be implemented by its subclasses. Abstract classes are used when you want to

provide some common functionality along with a set of methods that subclasses are required to implement.
    - In some programming languages, an interface is a type that only declares methods (no implementation). All methods in an interface are abstract by default. Interfaces specify what methods a class must implement but do not provide any code for those methods. In Python, interfaces are often represented using abstract base classes with only abstract methods, similar to how interfaces work in languages like Java.
    - Key Differences:
        - Implementation: Abstract classes can have both abstract and concrete methods, while interfaces typically only have abstract methods. Multiple Inheritance: A class can implement multiple interfaces, allowing for multiple inheritance of types. However, a class can only inherit from one abstract class (in languages that do not support multiple inheritance). Usage: Use abstract classes when you need to share common code among multiple related classes. Use interfaces when you want to specify a contract for classes without dictating any common implementation.

- What is polymorphism in object-oriented programming, and how is it implemented in Python?
    - Polymorphism is a fundamental concept in object-oriented programming that allows objects of different classes to be treated as objects of a common superclass. It enables a single interface to represent different underlying data types and methods, which can lead to more flexible and reusable code.
    - Polymorphism can be achieved in two main ways:
        - Method Overriding: Subclasses provide specific implementations of methods that are defined in their superclass. Method Overloading: Multiple methods with the same name but different parameters (not natively supported in Python but can be simulated).
    - In Python, polymorphism is typically implemented through method overriding and the use of interfaces (abstract base classes).

Design Patterns

Singleton

When an object can only be instantiated once

Prototype (Clone)

Another way to have inheritance

Builder

Create an object and let methods add values to the attributes instead of all at once

Facade

Simplified API that the end user doesn't have to see

Proxy

Substitute.

5 Design Pattern

1. Strategy Pattern
    - sounds like polymorphism
2. decorator pattern
    - open to extension / closed to modification
    - wrap a class into another class
3. Observer Pattern
    - notify interested party that something has happened
4. Singleton Pattern
    - Single object and make sure there's a single object
5. Facade Pattern
    - Different libraries and frameworks

Others:

1. Model-View-Controller (MVC): The MVC pattern is used for designing user interfaces. It separates the application into three distinct components: the model (data layer), the view (UI layer), and the controller (business logic). This separation of concerns makes it easier to maintain the application and make changes as needed.
2. Singleton: The singleton pattern is used when you need to have only one instance of a class in an application. This ensures that all parts of the application use the same instance of the class and prevents multiple instances from being created.
3. Observer: The observer pattern is used to allow objects to observe other objects and be notified of any changes in their state. This is useful for maintaining consistency across an application or for building event-driven systems.
4. Factory: The factory pattern is used for creating objects. It allows the application to delegate object creation to a separate class and makes it easier to maintain the code by hiding implementation details from the client code.
5. Adapter: The adapter pattern is used for converting one interface into another. This allows existing classes with incompatible interfaces to work together without having to rewrite existing code.
6. Strategy: The strategy pattern is used for selecting algorithms at runtime. It is useful for creating extensible applications that can easily switch between different algorithms or behaviors.
7. Command: The command pattern is used for encapsulating actions as objects. This makes it easier to undo, redo, and log actions in an application.