Object-Oriented Programming (OOP) concepts in Python and Java

1. Encapsulation – Restricting Access to Internal Data

- **What?** Encapsulation involves wrapping data (fields) and methods (functions) within a class and restricting access to some components. This is often done by making attributes private and providing public getter/setter methods.

- **Why?** Encapsulation protects an object's internal state by preventing unauthorized or improper changes. It enforces data integrity and abstraction, which makes it easier to modify and maintain.

```
// Encapsulation example in Java

class Employee {

    private String name;

    private double salary;

    public Employee(String name, double salary) {

        this.name = name;

        this.salary = salary;

    }

    // Public getter and setter methods

    public String getName() {

        return name;

    }

    public double getSalary() {

        return salary;

    }
```

```
    public void setSalary(double salary) {
        if (salary > 0) {
            this.salary = salary;
        } else {
            System.out.println("Invalid salary!");
        }
    }
}
public class Main {
    public static void main(String[] args) {
        Employee emp = new Employee("John", 50000);
        System.out.println("Initial Salary: " + emp.getSalary());
        emp.setSalary(60000);  // Valid
        System.out.println("Updated Salary: " + emp.getSalary());
        emp.setSalary(-100);    // Invalid, restricted by
encapsulation
    }
}
```

 # Encapsulation example in Python

```
class Employee:
    def __init__(self, name, salary):
        self.__name = name  # private attribute (convention, not enforced)
        self.__salary = salary
    # Getter and setter methods
    def get_name(self):
        return self.__name
```

```
    def get_salary(self):
        return self.__salary

    def set_salary(self, salary):
        if salary > 0:
            self.__salary = salary
        else:
            print("Invalid salary!")

# Testing encapsulation
emp = Employee("John", 50000)
print("Initial Salary:", emp.get_salary()
emp.set_salary(60000)  # Valid update
print("Updated Salary:", emp.get_salary())
emp.set_salary(-100)  # Invalid, rejected by setter method
```

2. Inheritance – Reusing Code from Parent Classes ("Is-a" Relationship)

- **What?** Inheritance allows a new class (subclass) to inherit attributes and methods from an existing class (superclass). This promotes code reuse and hierarchical classification.

- **Why?** By using inheritance, you avoid code duplication. Instead of rewriting shared functionality in multiple classes, you place it in a base class and let subclasses inherit and potentially override or extend this functionality.

```
// Inheritance example in Java
class Vehicle {
    public String brand;

   public Vehicle(String brand){
     this.brand = brand;
   }
    public void startEngine() {
        System.out.println("Engine started");
    }
}

class Car extends Vehicle {
    public String modelName;
    public Car(String brand, String modelName){
         super(brand);
         this.moelName = modelName;
    }

    public void displayInfo() {
        System.out.println("Brand: " + brand + ", Model: " +
modelName);
    }
}
public class Main {
    public static void main(String[] args) {
        Car car = new Car("Ford", "Mustang");
```

```
        car.startEngine();
        car.displayInfo();
    }
}
```

```
# Inheritance example in Python
class Vehicle:
    def __init__(self, brand):
        self.brand = brand

    def start_engine(self):
        print("Engine started")

class Car(Vehicle):
    def __init__(self, brand, model):
        super().__init__(brand)
        self.model = model

    def display_info(self):
        print(f"Brand: {self.brand}, Model: {self.model}")

# Testing inheritance
car = Car("Ford", "Mustang")
car.start_engine()
car.display_info()
```

3. Polymorphism – Using a Common Interface to Interact with Objects of Different Types

- **What?** Polymorphism allows objects of different classes to be treated as objects of a common superclass. It allows a single interface to be used with objects of different types, typically through method overriding or overloading.

- **Why?** Polymorphism increases flexibility by allowing the same method to behave differently based on the object it is called on. This leads to more generic, reusable code.

```
// Polymorphism example in Java

class Employee {

    public void work() {

        System.out.println("Employee is working");

    }

}

class Manager extends Employee {

    @Override

    public void work() {

        System.out.println("Manager is managing the team");

    }

}

class Developer extends Employee {

    @Override

    public void work() {

        System.out.println("Developer is writing code");

    }

}
```

```
// Testing polymorphism
public class Main {
    public static void assignWork(Employee employee) {
        employee.work();
    }
    public static void main(String[] args) {
        Employee emp1 = new Manager();
        Employee emp2 = new Developer();
        assignWork(emp1);// Outputs: Manager is managing the team
        assignWork(emp2);  // Outputs: Developer is writing code
    }
}
```

 # Polymorphism example in Python

```
class Employee:
    def work(self):
        print("Employee is working")

class Manager(Employee):
    def work(self):
        print("Manager is managing the team")
class Developer(Employee):
    def work(self):
        print("Developer is writing code")

# Testing polymorphism
def assign_work(employee):
    employee.work()
```

```
emp1 = Manager()
emp2 = Developer()
assign_work(emp1)  # Outputs: Manager is managing the team
assign_work(emp2)  # Outputs: Developer is writing code
```

4. Abstraction – Hiding Complex Implementations and Exposing Simple Interfaces

- **What?** Abstraction hides the complexity of an object's internal workings and only exposes the necessary details through an interface or an abstract class. This helps in designing clean interfaces that are easy to understand.

- **Why?** Abstraction simplifies interactions by providing a clear, minimal interface for the functionality while hiding the complex inner workings. It allows developers to focus on *what* an object does rather than *how* it does it.

```
// Abstraction example in Java
abstract class Report {
    protected String title;
    public Report(String title) {
            this.title = title;
    }
    public abstract void generate(); // Generates the report content
}
class PDFReport extends Report {
    public PDFReport(String title) { super(title); }
    public void generate() {
            System.out.println("Generating PDF: " + title); }
    }
```

```
class HTMLReport extends Report {
    public HTMLReport(String title) { super(title); }
    public void generate() {
      System.out.println("Generating HTML: " + title);
    }
}
class WordReport extends Report {
    public WordReport(String title) { super(title); }
    public void generate() {
      System.out.println("Generating Word: " + title);
    }
}
class ReportFactory {
    public static Report createReport(String type, String title)
{
        if (type.equalsIgnoreCase("pdf")) {
            return new PDFReport(title);
        } else if (type.equalsIgnoreCase("html")) {
            return new HTMLReport(title);
        } else if (type.equalsIgnoreCase("word")) {
            return new WordReport(title);
        } else {
            throw new IllegalArgumentException("Unknown type");
        }
    }
    public static String printReportTitle(Report report) {
        return "Report Title: " + report.title;
```

```
    }
}
public class Main {
    public static void main(String[] args) {
        Report report = ReportFactory.createReport("word",
"Quarterly Review");
        report. generate();  // Outputs: Generating Word: Quarterly
Review
        String reportTitle = ReportFactory.printReportTitle(report);
        System.out.println(reportTitle);  // Outputs: Report
Title: Quarterly Review
    }
}
```

 # Abstraction example in Python

```
from abc import ABC, abstractmethod
class Report(ABC):
    def __init__(self, title):
        self.title = title
    @abstractmethod
    def generate(self):
        pass


class PDFReport(Report):
    def generate(self):
        print(f"Generating PDF: {self.title}")


class HTMLReport(Report):
    def generate(self):
```

```
        print(f"Generating HTML: {self.title}")

class WordReport(Report):
    def generate(self):
        print(f"Generating Word: {self.title}")

# Factory method returning different report types
def create_report(report_type, title):
    report_type = report_type.lower()
    if report_type == "pdf":
        return PDFReport(title)
    elif report_type == "html":
        return HTMLReport(title)
    elif report_type == "word":
        return WordReport(title)
    else:
        raise ValueError("Unknown report type")

# Method to return report title
def print_report_title(report):
    return f"Report Title: {report.title}"

report = create_report("word", "Quarterly Review")
report.generate()  # Outputs: Generating Word: Quarterly Review

report_title = print_report_title(report)
print(report_title)  # Outputs: Report Title: Quarterly Review
```

5. Association – Defining Relationships Between Classes

What? Association is a relationship between two classes where one class uses or interacts with another, but without ownership. There is no strong dependency, and both classes can exist independently.

Why? Association models real-world relationships, where objects can interact without implying any ownership or containment. It's useful for representing situations like a customer placing an order, without the customer being contained by the order.

// Association example in Java

```
class Customer {
    private String name;
    public Customer(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }
}
class Order {
    private int orderId;
    public Order(int orderId) {
        this.orderId = orderId;
    }
    public void placeOrder(Customer customer) {
        System.out.println("Order placed by " +
customer.getName() + " with order ID " + orderId);
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Customer customer = new Customer("Alice");
        Order order = new Order(123);
        order.placeOrder(customer);
    }
}
```

 # Association example in Python

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

    def get_name(self):
        return self.name

class Order:
    def __init__(self, order_id):
        self.order_id = order_id

    def place_order(self, customer):
        print(f"Order placed by {customer.get_name()} with order
ID {self.order_id}")

# Testing association
customer = Customer("Alice")
order = Order(123)
order.place_order(customer)
```

6. Aggregation – "Has-a" Relationship Where Objects Can Exist Independently

What? Aggregation is a relationship where one object contains or is composed of other objects, but the contained objects can exist independently outside the container. In a library, the Library contains many Books. However, if the library is closed or destroyed, the books themselves can still exist and function independently.

Why? Aggregation allows for complex relationships between objects without tightly coupling them. This provides flexibility and allows for modularity, where objects can be composed and recomposed without tightly binding their lifecycles.

```
// Aggregation Example in Java
class Book {
    private String title;
    public Book(String title) {
        this.title = title;
    }

    public String getTitle() {
        return title;
    }
}
// Class representing a Library (aggregation relationship with Book)
class Library {
    private List<Book> books;
    public Library() {
        this.books = new ArrayList<>();
```

```
    }
    public void addBook(Book book) {
        books.add(book);
    }

    public void showBooks() {
        for (Book book : books) {
            System.out.println("Book: " + book.getTitle());
        }
    }
}
public class Main {
    public static void main(String[] args) {
        // Creating books
        Book book1 = new Book("Effective Java");
        Book book2 = new Book("Clean Code");

        // Creating library and adding books
        Library library = new Library();
        library.addBook(book1);
        library.addBook(book2);
        library.showBooks(); //Outputs: Effective Java, Clean Code

    }
}
```

```
# Aggregation example in Python
class Book:
    def __init__(self, title):
        self.title = title

    def get_title(self):
        return self.title
# Class representing a Library (aggregation relationship with Book)
class Library:
    def __init__(self):
        self.books = []

    def add_book(self, book):
        self.books.append(book)

    def show_books(self):
        for book in self.books:
            print(f"Book: {book.get_title()}")

# Client code
book1 = Book("Effective Java")
book2 = Book("Clean Code")
library = Library()
library.add_book(book1)
library.add_book(book2)
library.show_books()  # Outputs: Effective Java, Clean Code
```

7. Composition – Strong "Has-a" Relationship Where Objects Depend on Each Other

What? Composition represents a strong "has-a" relationship where the lifetime of the contained object depends on the container. If the container object is destroyed, the contained objects are also destroyed. For example, a car "has-a" engine, and if the car is destroyed, the engine is also destroyed.

Why? Composition ensures that objects cannot exist independently, which models real-world relationships more tightly than aggregation. It's useful when one object is a vital part of another and shouldn't exist without it.

```
// Aggregation Example in Java

// Composition example in Java

class Engine {

    public void start() {

        System.out.println("Engine started");

    }

}

class Car {

    private Engine engine;

    public Car() {

        engine = new Engine();  // Composition: Car "has-a" Engine

    }

    public void startCar() {

        engine.start();

        System.out.println("Car is running");

    }

}
```

```
public class Main {
    public static void main(String[] args) {
        Car car = new Car();
        car.startCar();  // Outputs: Engine started, Car is running
    }
}
```

```
# Composition example in Python
class Engine:
    def start(self):
        print("Engine started")

class Car:
    def __init__(self):
        self.engine = Engine()  # Composition: Car "has-a" Engine

    def start_car(self):
        self.engine.start()
        print("Car is running")

# Testing composition
car = Car()
car.start_car()  # Outputs: Engine started, Car is running
```