**I. OOP Theory Questions**

**1. Core OOP Concepts What are the four main principles of object-oriented programming? Explain each with examples.**

**a. Đóng gói (Encapsulation)**: Gói dữ liệu và phương thức thao tác trên dữ liệu trong một đơn vị duy nhất (class). Ví dụ với class Fraction:

* **Đóng gói (Encapsulation)** bao gồm:
  + **Thuộc tính (Attributes/Fields)** – Biến lưu trữ dữ liệu, thường đặt là private để bảo vệ dữ liệu.
  + **Constructor (Hàm khởi tạo)** – Dùng để khởi tạo đối tượng với giá trị ban đầu.
  + **Property (Thuộc tính truy cập)** – Cung cấp cách đọc/ghi giá trị an toàn, có thể thêm logic kiểm tra.
  + **Phương thức (Methods)** – Thao tác trên dữ liệu, có thể đặt private để chỉ sử dụng nội bộ class.
* **Mục đích:** bảo vệ dữ liệu và ngăn chặn việc thay đổi dữ liệu trực tiếp từ bên ngoài, giúp code an toàn và dễ bảo trì hơn.
* **Lợi ích của đóng gói đối với bảo mật và bảo trì mã nguồn:**
  + Ngăn chặn truy cập trực tiếp vào dữ liệu bên trong class, giúp tránh lỗi hoặc thao tác sai dữ liệu.
  + Giúp bảo trì dễ dàng hơn, khi thay đổi nội bộ không ảnh hưởng đến phần còn lại của chương trình.

A computer screen shot of a program code

AI-generated content may be incorrect.A computer screen shot of a program code

AI-generated content may be incorrect.

**b. Trừu tượng hóa (Abstraction)**

* **Định nghĩa**: Trừu tượng hóa là việc ẩn đi các chi tiết triển khai không cần thiết và chỉ cung cấp những gì quan trọng cho người dùng.
* **Mục đích**: Giúp đơn giản hóa việc sử dụng class và đảm bảo tính linh hoạt trong hệ thống.
* **Cách triển khai**: Thường được thực hiện thông qua **lớp trừu tượng (abstract class)** và **giao diện (interface)**.
* **Khác biệt giữa trừu tượng hóa và đóng gói:**
  + Đóng gói bảo vệ dữ liệu bên trong class, hạn chế quyền truy cập.
  + Trừu tượng hóa ẩn đi các chi tiết triển khai phức tạp, chỉ cung cấp giao diện cần thiết.

A screen shot of a computer

AI-generated content may be incorrect.A computer screen shot of a program code

AI-generated content may be incorrect.

* FractionBase là lớp trừu tượng với phương thức Display() chưa có triển khai.
* Fraction kế thừa từ FractionBase và cài đặt phương thức Display().

**c. Kế thừa (Inheritance)**

* **Định nghĩa**: Kế thừa là cơ chế cho phép một class mới **tái sử dụng** các thuộc tính và phương thức từ một class có sẵn.
* **Mục đích**: Giúp giảm trùng lặp mã và tổ chức hệ thống theo một cấu trúc hợp lý hơn.
* **Cách triển khai**: Sử dụng dấu :
* **Ưu điểm của kế thừa:**
  + Tái sử dụng mã nguồn, giảm lặp code.
  + Dễ dàng mở rộng tính năng.
  + Hỗ trợ đa hình, cho phép ghi đè phương thức để thay đổi hành vi.
* **Rủi ro của kế thừa:**
  + Nếu không thiết kế đúng, có thể gây phụ thuộc chặt chẽ giữa các class, khó bảo trì.
  + Nếu lạm dụng kế thừa, có thể dẫn đến hệ thống phức tạp và khó kiểm soát.
  + **Vấn đề "Lớp cơ sở mong manh"** – Thay đổi lớp cha có thể làm hỏng lớp con.
  + **Sử dụng kế thừa không hợp lý** – Đôi khi nên dùng **thành phần (Composition)** thay vì kế thừa.

A computer screen shot of a program

AI-generated content may be incorrect.

**d. Đa hình (Polymorphism)**

* **Định nghĩa**: Đa hình cho phép phương thức có thể có nhiều hình thức khác nhau, gồm:
  + **Ghi đè phương thức (Method Overriding)**: Khi một lớp con thay đổi cách triển khai phương thức từ lớp cha.
  + **Nạp chồng phương thức (Method Overloading)**: Khi có nhiều phương thức cùng tên nhưng khác tham số.
* **Mục đích**: Giúp mã nguồn linh hoạt hơn và dễ mở rộng.
* **Phân biệt đa hình thời gian biên dịch và đa hình thời gian chạy:**
  + **Đa hình thời gian biên dịch (Compile-time polymorphism):** Thực hiện bằng nạp chồng phương thức (method overloading).
  + **Đa hình thời gian chạy (Runtime polymorphism):** Thực hiện bằng ghi đè phương thức (method overriding).

A black background with text

AI-generated content may be incorrect.A computer code with text

AI-generated content may be incorrect.

* Fraction có phương thức ToString(), lớp MixedFraction ghi đè (override) phương thức này để hiển thị theo cách khác.

**2. Tính đóng gói giúp cải thiện bảo mật và bảo trì mã như thế nào?**

**Đóng gói (Encapsulation)** là nguyên tắc OOP giúp **ẩn các chi tiết bên trong** của một đối tượng và **hạn chế quyền truy cập trực tiếp** vào một số thành phần của nó.

* **Cải thiện bảo mật:**
  + Ngăn chặn **truy cập trái phép** vào dữ liệu bằng cách sử dụng các biến private và chỉ cho phép thao tác thông qua các phương thức public.
  + Đảm bảo **toàn vẹn dữ liệu** bằng cách xác thực dữ liệu trước khi thay đổi.
* **Cải thiện bảo trì:**
  + **Giúp mã nguồn dễ sửa đổi**, vì một phần của hệ thống có thể thay đổi mà không ảnh hưởng đến phần khác.
  + **Giảm độ phức tạp**, giúp người dùng chỉ quan tâm đến cách sử dụng mà không cần biết cách nó hoạt động bên trong.
  + **Tăng tính tái sử dụng**, vì các lớp có hành vi được đóng gói có thể dễ dàng sử dụng lại

**3. Trừu tượng trong OOP là gì? Nó khác gì so với đóng gói?**

**Trừu tượng (Abstraction)** là nguyên tắc **ẩn đi các chi tiết triển khai** và chỉ hiển thị các chức năng quan trọng cho người dùng.

* **Ví dụ về trừu tượng:**
  + Lớp Car có phương thức Drive(), nhưng người dùng không cần biết động cơ hoạt động như thế nào.
* **So sánh Trừu tượng và Đóng gói:**

| **Đặc điểm** | **Đóng gói** | **Trừu tượng** |
| --- | --- | --- |
| **Mục đích** | Ẩn dữ liệu để bảo vệ khỏi thay đổi ngoài ý muốn. | Ẩn chi tiết triển khai để chỉ hiển thị chức năng quan trọng. |
| **Cách thực hiện** | Dùng biến private với các phương thức public. | Dùng **lớp trừu tượng (abstract class) hoặc giao diện (interface)**. |
| **Ví dụ** | Biến số dư tài khoản ngân hàng là private, chỉ thay đổi qua phương thức. | Lớp Vehicle có phương thức Start(), nhưng không quan tâm nó hoạt động như thế nào. |

**4. Đa hình là gì? Sự khác nhau giữa đa hình tĩnh và đa hình động?**

**Đa hình (Polymorphism)** nghĩa là **"nhiều hình dạng"**, cho phép một giao diện duy nhất đại diện cho nhiều kiểu hành vi khác nhau.

* **Hai loại đa hình:**

|  |  |
| --- | --- |
| **Overloading** | **Overriding** |
| Thể hiện đa hình tại **compile time** | Thể hiện đa hình tại **runtime** |
| Thêm hành vi cho phương thức | Thay đổi hành vi hiện tại của phương thức |
| Có thể khác nhau về số lượng và kiểu dữ liệu của tham số | Số lượng và kiểu dữ liệu của tham số phải giống nhau |
| Xảy ra trong cùng một class | Xảy ra ở 2 class có quan hệ kế thừa |

**5. Sự khác nhau giữa Interface và Abstract Class?**

Cả **giao diện (Interface)** và **lớp trừu tượng (Abstract Class)** đều hỗ trợ trừu tượng, nhưng có điểm khác biệt:

| **Đặc điểm** | **Interface** | **Abstract Class** |
| --- | --- | --- |
| **Mục đích** | Xác định hợp đồng (contract) mà các lớp phải tuân theo. | Xác định các hành vi chung và có thể cung cấp triển khai mặc định. |
| **Phương thức** | Không có thân hàm (trước C# 8). | Có thể có cả phương thức trừu tượng và phương thức có sẵn triển khai. |
| **Trường dữ liệu** | Không có biến thành viên. | Có thể có biến thành viên. |
| **Đa kế thừa** | Một lớp có thể triển khai nhiều interface. | Một lớp chỉ có thể kế thừa một abstract class. |
| **Khi nào dùng?** | Khi các lớp không liên quan cần có cùng hành vi. | Khi các lớp có quan hệ cha-con với nhau. |

**Khi nào nên dùng Interface :** **Khi nhiều lớp cần cùng một hành vi nhưng không có chung quan hệ kế thừa.**

* Cần cung cấp các chức năng chung cho các lớp không liên quan.
* Cần nhóm các đối tượng dựa trên các hành vi phổ biến.
* Cần sử dụng đa hình vì một lớp có thể thực hiện nhiều interfaces.
* Cần tạo các thành phần (components) được ghép lỏng lẻo, dễ bảo trì và dùng như 1 plugin vì việc implement cho interface được tách biệt với nó.

**Khi nào nên dùng Abstract Class : Khi các lớp có chung logic cơ bản và có thể chia sẻ mã nguồn.**

* Cần sử dụng kế thừa.
* Cần cung cấp các phương thức mặc định cũng như các phương thức phổ biến mà nhiều lớp dẫn xuất có thể thực thi và ghi đè.
* Cần tạo nhiều phiên bản cho các thành phần (components). Bạn có thể thêm các thuộc tính và phương thức vào Abstract Class mà không vi phạm mã và tất cả các lớp kế thừa được tự động cập nhật với thay đổi.

|  |  |
| --- | --- |
| **Abstract class** | **Interface** |
| Có các phương thức trừu tượng và không trừu tượng | Chỉ có duy nhất phương thức trừu tượng |
| Không hỗ trợ đa kế thừa | Hỗ trợ đa kế thừa |
| Có thể tồn tại các biến final và non-final, static và non-static | Chỉ có các biến static và final |
| Có thể extents (kế thừa) interface | Không thể kế thừa lớp abstract |
| Sử dụng từ khóa “abstract” để khai báo lớp trừu tượng | Sử dụng từ khóa “interface” để khai báo giao diện |
| Một lớp trừu tượng có thể extents một lớp Java khác và thực hiện nhiều interface khác nhau | Một interface chỉ có thể implements (cài đặt) nhiều interface khác |
| Có thể kế thừa bằng cách sử dụng từ khóa “extends” | Các lớp triển khai interface bằng cách sử dụng từ khóa “implements” |
| Một lớp trừu tượng có phạm vi truy cập ở dạng private, protected,… | Các phương thức, property của interface đều được mặc định ở dạng public |
| Có thể định nghĩa thân phương thức và property. | Chỉ có thể khai báo, không thể định nghĩa code xử lý. |

**2. Advanced OOP Concepts**

**1. Đa kế thừa là gì? Nó được xử lý như thế nào trong các ngôn ngữ không hỗ trợ trực tiếp?**

**Đa kế thừa (Multiple Inheritance)** là tính năng trong lập trình hướng đối tượng cho phép một lớp kế thừa từ nhiều lớp cha. Điều này giúp lớp con có thể thừa hưởng thuộc tính và hành vi từ nhiều lớp cha.

Tuy nhiên, đa kế thừa có thể gây ra vấn đề như **vấn đề kim cương (diamond problem)**, khi một phương thức được kế thừa từ nhiều lớp cha gây ra sự mơ hồ trong việc xác định phiên bản nào sẽ được sử dụng.

Các ngôn ngữ không hỗ trợ đa kế thừa trực tiếp (ví dụ: **Java, C#**) xử lý bằng cách:

* **Giao diện (Interfaces)**: Một lớp có thể triển khai nhiều giao diện thay vì kế thừa nhiều lớp.
* **Thành phần (Has-a relationship)**: Một lớp có thể chứa các đối tượng của lớp khác và ủy quyền phương thức thay vì kế thừa.
* **Lớp trừu tượng và ghi đè phương thức**: Sử dụng lớp trừu tượng chứa các phương thức chung và cho phép lớp con kế thừa để mở rộng.

**2. Phương thức ảo là gì? Nó liên quan thế nào đến ghi đè phương thức?**

**Phương thức ảo (Virtual Method)** là phương thức được khai báo trong lớp cha và được thiết kế để các lớp con có thể ghi đè.

* Trong **C++**, phương thức được đánh dấu virtual.
* Trong **C#**, phải khai báo virtual trong lớp cha và override trong lớp con.
* Trong **Java**, tất cả các phương thức mặc định là ảo trừ khi được đánh dấu final.

**Ghi đè phương thức (Method Overriding)** xảy ra khi lớp con cung cấp một phiên bản riêng của phương thức đã có trong lớp cha. Điều này hỗ trợ **đa hình (polymorphism)**, cho phép lớp con thay đổi hành vi của lớp cha.

**3. Mối quan hệ "is-a" và "has-a" trong OOP**

* **"Is-a" (Là một) - Quan hệ kế thừa**
  + Một lớp con kế thừa từ một lớp cha.
  + Ví dụ: Chó **là một** loại Động vật.
  + Triển khai bằng kế thừa (inheritance).
* **"Has-a" (Có một) - Quan hệ thành phần**
  + Một lớp chứa một đối tượng của lớp khác.
  + Ví dụ: Xe hơi **có một** Động cơ.
  + Triển khai bằng cách sử dụng thuộc tính (fields/properties).

**4. Nguyên tắc SOLID trong OOP? Ví dụ thực tế cho từng nguyên tắc?**

Các nguyên tắc **SOLID** giúp mã nguồn dễ bảo trì và mở rộng hơn.

1. **S - Single Responsibility Principle (SRP) - Nguyên tắc đơn nhiệm**
   * **Mỗi lớp chỉ nên có một lý do để thay đổi.**
   * **Ví dụ:** Trong hệ thống thương mại điện tử, lớp Order không nên xử lý việc in hóa đơn. Thay vào đó, nên có lớp InvoicePrinter riêng.
2. **O - Open/Closed Principle (OCP) - Nguyên tắc mở/đóng**
   * **Lớp nên mở để mở rộng nhưng đóng để sửa đổi.**
   * **Ví dụ:** Hệ thống thanh toán nên cho phép thêm các phương thức mới (PayPal, Stripe) mà không cần chỉnh sửa mã cũ. Điều này có thể thực hiện bằng cách sử dụng interface.
3. **L - Liskov Substitution Principle (LSP) - Nguyên tắc thay thế Liskov**
   * **Lớp con phải có thể thay thế lớp cha mà không làm thay đổi hành vi mong đợi của chương trình.**
   * **Ví dụ:** Một lớp Hình chữ nhật không nên bị thay thế bằng Hình vuông nếu điều này làm sai lệch cách thức hoạt động của lớp cha.
4. **I - Interface Segregation Principle (ISP) - Nguyên tắc phân tách giao diện**
   * **Không nên ép buộc lớp phải triển khai những phương thức mà chúng không cần.**
   * **Ví dụ:** Thay vì có giao diện Worker với Work() và EatLunch(), nên tách thành IWorker (có Work()) và IEater (có EatLunch()).
5. **D - Dependency Inversion Principle (DIP) - Nguyên tắc đảo ngược phụ thuộc**
   * **Lớp cao cấp không nên phụ thuộc vào lớp thấp cấp, cả hai nên phụ thuộc vào abstraction.**
   * **Ví dụ:** DatabaseService nên phụ thuộc vào IDatabase (một interface) thay vì SQLDatabase, giúp dễ dàng thay đổi cơ sở dữ liệu.

**5. Nguyên tắc thay thế Liskov (Liskov Substitution Principle - LSP) là gì? Tại sao nó quan trọng?**

Nguyên tắc **Liskov Substitution Principle (LSP)** nói rằng một lớp con có thể thay thế lớp cha **mà không làm thay đổi hành vi mong đợi của chương trình**.

**Tại sao quan trọng?**

* Giúp đảm bảo tính nhất quán khi sử dụng kế thừa.
* Ngăn chặn lỗi bất ngờ khi sử dụng đa hình.

A screen shot of a computer program

AI-generated content may be incorrect.A screenshot of a computer program

AI-generated content may be incorrect.

* Ở đây, lớp Square vi phạm LSP vì khi đặt Width, nó cũng thay đổi Height, làm sai lệch cách hoạt động của Rectangle.
* Giải pháp: Không nên coi Square là một Rectangle, mà nên dùng hai lớp riêng biệt.

**6. Cách triển khai Dependency Injection trong OOP?**

**Dependency Injection (DI)** là một mô hình thiết kế giúp tách rời phụ thuộc bằng cách truyền đối tượng phụ thuộc từ bên ngoài vào thay vì khởi tạo bên trong lớp. Điều này giúp mã dễ mở rộng và kiểm thử hơn.

Có ba cách chính để triển khai DI:

1. **Constructor Injection** (tốt nhất)
2. **Property Injection**
3. **Method Injection**

**3. OOP Design Patterns**

**1. Design Pattern là gì? Tại sao chúng quan trọng?**

**Design Pattern** là các giải pháp đã được kiểm chứng để giải quyết các vấn đề thiết kế phần mềm phổ biến trong lập trình hướng đối tượng (OOP).

* **Tại sao Design Pattern quan trọng?**
* **Tái sử dụng mã**: Giảm thiểu việc viết lại mã từ đầu.
* **Tăng tính linh hoạt**: Giúp phần mềm dễ bảo trì, mở rộng.
* **Chuẩn hóa thiết kế**: Giúp các nhóm phát triển dễ dàng hiểu và làm việc với mã nguồn.
* **Giảm lỗi thiết kế**: Tránh các sai lầm phổ biến trong lập trình.
* **Các nhóm Design Pattern phổ biến:**
  + **Creational (Mẫu tạo)** – Quản lý cách tạo đối tượng (Singleton, Factory, Builder...).
  + **Structural (Mẫu cấu trúc)** – Tổ chức mối quan hệ giữa các lớp (Adapter, Decorator, Proxy...).
  + **Behavioral (Mẫu hành vi)** – Quản lý cách các đối tượng tương tác với nhau (Observer, Strategy, Command...).
* **Đảm bảo rằng 1 lớp (class) sẽ chỉ có 1 instance duy nhất**: Sẽ có 1 số trường hợp mà bạn cần kiểm soát việc truy cập đến các tài nguyên dùng chung ví dụ như database hay 1 file nào đó; lúc này bạn cần kiểm soát được số lượng instance mà 1 class đó có.
* **Cung cấp 1 điểm truy cập global đến instance đó**: Biến global (toàn cục) thường được sử dụng để lưu trữ 1 số đối tượng thiết yếu. Mặc dù nó rất là tiện dụng, nhưng chúng cũng rất không an toàn vì bất cứ đoạn code nào trong chương trình cũng có thể ghi đè nội dung của những biến đó khiến cho ứng dụng của chúng ta bị crash. Singleton cũng giống như biến toàn cục, nó cho phép bạn truy cập đến 1 số object ở bất kỳ đâu trong chương trình, tuy nhiên nó cũng bảo vệ instance đó tránh khỏi việc bị ghi đè bởi code khác.

**Để biến một class thành Singleton, cần đảm bảo rằng:**

* Định nghĩa một attribute là private static và đó là thể hiện duy nhất của class này
* Định nghĩa public static getInstance() dùng để khởi tạo đối tượng (hàm accessor)
* Thực hiện lazy-init trong hàm accessor (chỉ khi gọi mới khởi tạo thể hiện)
* Constructor (hoặc các constructor) là private hay protected, vì bạn không muốn client tạo nhiều thể hiện
* Client chỉ có thể gọi hàm accessor khi muốn có thể hiện của class

A screenshot of a computer program

AI-generated content may be incorrect. A screenshot of a computer program

AI-generated content may be incorrect.

**3. Mẫu Factory là gì? Nó cải thiện việc tạo đối tượng như thế nào?**

**Factory Pattern** là mẫu thiết kế giúp **tạo đối tượng mà không cần chỉ rõ lớp cụ thể**.

Factory Pattern giúp **cải thiện quá trình tạo đối tượng** theo ba cách chính:

**a. Tách rời logic khởi tạo khỏi code sử dụng đối tượng**

* Nếu không dùng Factory Pattern, code khởi tạo đối tượng sẽ xuất hiện khắp nơi trong chương trình. Điều này làm cho mã nguồn khó bảo trì.
* Với Factory, việc khởi tạo được tập trung vào **một nơi duy nhất**.

A screenshot of a computer

AI-generated content may be incorrect.

**b. Đơn giản hóa việc mở rộng (Dễ thêm loại đối tượng mới mà không sửa code cũ)**

Khi muốn thêm một loại phương tiện mới (ví dụ: Bus), **chỉ cần sửa Factory**, không cần thay đổi nơi sử dụng.A screenshot of a computer program

AI-generated content may be incorrect.

**3. Tăng tính linh hoạt bằng cách kết hợp với Dependency Injection**

* Factory giúp tạo đối tượng linh hoạt hơn bằng cách kết hợp với **Dependency Injection (DI)**.
* Ví dụ, thay vì tạo trực tiếp đối tượng, ta có thể sử dụng DI để **cấu hình đối tượng từ bên ngoài**, giúp code dễ kiểm thử hơn.

A screenshot of a computer program

AI-generated content may be incorrect.

**4. Mẫu Strategy là gì? Khi nào nên sử dụng?**

**Strategy Pattern** cho phép thay đổi thuật toán **mà không cần sửa đổi mã nguồn chính**.

* **Khi nào dùng Strategy Pattern?**
* Khi một lớp có **nhiều cách thực hiện hành vi khác nhau**.
* Khi cần thay đổi thuật toán **mà không làm ảnh hưởng đến code cũ**.

**Ứng dụng thực tế:**

* **Lựa chọn thuật toán sắp xếp (QuickSort, MergeSort, BubbleSort).**
* **Chọn phương thức thanh toán trong hệ thống thương mại điện tử.**

A screen shot of a computer program

AI-generated content may be incorrect.

**5. Mẫu Observer là gì? Ví dụ thực tế?**

**Observer Pattern** cho phép một đối tượng (Observable) thông báo cho nhiều đối tượng khác (Observers) khi có thay đổi.

* **Ứng dụng của Observer Pattern:**
  + **Hệ thống thông báo (Notification System).**
  + **Cập nhật giao diện người dùng khi dữ liệu thay đổi.**
  + **Mô hình Publish-Subscribe (Event-driven programming).**

A screen shot of a computer program

AI-generated content may be incorrect.

**6. So sánh Decorator và Proxy Pattern**

| **Đặc điểm** | **Decorator** | **Proxy** |
| --- | --- | --- |
| **Mục đích** | Mở rộng chức năng của một đối tượng | Kiểm soát truy cập vào đối tượng |
| **Thay đổi hành vi** | Có thể thêm nhiều lớp decorator để mở rộng đối tượng | Không thay đổi hành vi chính, chỉ kiểm soát truy cập |
| **Ví dụ** | Thêm logging, caching vào một dịch vụ | Kiểm soát truy cập từ xa, bảo mật |

A screenshot of a computer program

AI-generated content may be incorrect.

**Ứng dụng thực tế:**

* **Decorator:** Thêm tính năng caching, logging cho service.
* **Proxy:** Bảo vệ truy cập vào database hoặc API từ xa.

1. **Object Lifecycle and Memory Management**

**1. Tạo và hủy đối tượng trong OOP**

**Trong lập trình hướng đối tượng (OOP), việc tạo và hủy đối tượng diễn ra theo cơ chế sau:**

a. Tạo đối tượng (Object Creation)

* + Khi một đối tượng được khởi tạo bằng từ khóa new (trong C#, Java), bộ nhớ sẽ được cấp phát trên heap.
  + Constructor (hàm khởi tạo) sẽ được gọi để thiết lập trạng thái ban đầu cho đối tượng.

b. Hủy đối tượng (Object Destruction)

* + Trong ngôn ngữ quản lý bộ nhớ thủ công như C++, đối tượng cần được giải phóng bằng cách gọi delete hoặc sử dụng destructor (~ClassName()).
  + Trong các ngôn ngữ có garbage collection như C# và Java, bộ thu gom rác (Garbage Collector - GC) sẽ tự động thu hồi bộ nhớ của các đối tượng không còn tham chiếu nào trỏ đến.

**2. Cách hoạt động của Garbage Collection trong C# và Java**

**Garbage Collection (GC) giúp quản lý bộ nhớ tự động, giảm thiểu rủi ro rò rỉ bộ nhớ. Cách hoạt động của GC như sau:**

a. Xác định đối tượng không còn được sử dụng

* GC kiểm tra các đối tượng không còn tham chiếu trong chương trình.

b. Đánh dấu và thu gom rác (Mark and Sweep)

* Marking: GC tìm tất cả các đối tượng có thể truy cập và đánh dấu chúng.
* Sweeping: Các đối tượng không được đánh dấu sẽ bị xóa khỏi bộ nhớ.

c. Nén (Compacting - áp dụng trong Java)

* + Sau khi thu gom, bộ nhớ sẽ được sắp xếp lại để giảm phân mảnh.

Trong C#, GC chia bộ nhớ heap thành 3 thế hệ:

* Gen 0: Các đối tượng mới tạo.
* Gen 1: Đối tượng sống lâu hơn.
* Gen 2: Đối tượng tồn tại lâu (thường là static hoặc global).

C# sử dụng GC.Collect() để kích hoạt GC thủ công, nhưng nên hạn chế sử dụng vì nó ảnh hưởng hiệu suất.

**3. Memory Leak là gì và cách phòng tránh trong OOP?**

**Memory Leak (rò rỉ bộ nhớ) xảy ra khi bộ nhớ cấp phát không được giải phóng đúng cách, khiến ứng dụng tiêu tốn tài nguyên không cần thiết. Nguyên nhân chính:**

* Giữ tham chiếu không cần thiết đến đối tượng.
* Sử dụng event handler mà không hủy đăng ký (-= eventHandler).
* Sử dụng tài nguyên không được giải phóng (IDisposable trong C#).

**Cách phòng tránh Memory Leak trong OOP:**

* Sử dụng Weak References (tránh giữ tham chiếu mạnh không cần thiết).
* Giải phóng tài nguyên với Dispose() và using (C#).
* Hủy đăng ký sự kiện khi không còn cần thiết.
* Kiểm tra Memory Profiler để phát hiện rò rỉ bộ nhớ.

**A screenshot of a computer

AI-generated content may be incorrect.**

**5. Principles and Best Practices**

**1. Tight Coupling vs. Loose Coupling**

**Tight Coupling (Liên kết chặt chẽ)**

* Khi các lớp trong hệ thống phụ thuộc trực tiếp vào nhau, thay đổi một lớp có thể ảnh hưởng đến nhiều lớp khác.
* Ví dụ: Một class gọi trực tiếp phương thức của một class khác thay vì thông qua interface hoặc abstraction.

**Loose Coupling (Liên kết lỏng lẻo)**

* Các thành phần ít phụ thuộc vào nhau, dễ mở rộng và bảo trì.
* Áp dụng Dependency Injection (DI) hoặc sử dụng Interface để giảm phụ thuộc trực tiếp.

**Tại sao Loose Coupling được ưu tiên?**

* Dễ bảo trì và mở rộng.
* Giảm sự phụ thuộc giữa các module, giúp dễ dàng thay đổi code.
* Hỗ trợ kiểm thử tốt hơn bằng cách thay thế các thành phần bằng mock object.

**2. Tại sao Composition thường được ưu tiên hơn Inheritance?**

* **Inheritance (Kế thừa)** tạo mối quan hệ chặt chẽ giữa lớp cha và lớp con, gây khó khăn khi thay đổi.
* **Composition (Thành phần)** cho phép thay thế và mở rộng chức năng mà không ảnh hưởng đến các class khác.

**Tại sao Composition tốt hơn?**

* Giảm sự phụ thuộc giữa các class.
* Dễ dàng thay đổi hành vi mà không ảnh hưởng đến cấu trúc hệ thống.
* Tránh vấn đề “kế thừa quá sâu” (deep inheritance).
* **3. Lợi ích của Immutable Objects trong OOP**
* **Immutable Object** là đối tượng mà trạng thái của nó không thể thay đổi sau khi được khởi tạo.
* **Lợi ích:**  
  An toàn khi sử dụng trong môi trường đa luồng (Thread-Safety).  
  Dễ bảo trì, ít lỗi liên quan đến trạng thái thay đổi.  
  Dễ dàng debug vì giá trị không thay đổi bất ngờ.

**4. Open/Closed Principle (Nguyên tắc Mở/Đóng)**

"Một module nên **mở** để mở rộng nhưng **đóng** để chỉnh sửa."

**Ý nghĩa:**

* Khi cần thay đổi hành vi của hệ thống, ta nên mở rộng bằng cách **thêm** code mới thay vì **sửa** code cũ.
* Áp dụng **interface, abstract class, và dependency injection** để tuân thủ nguyên tắc này.

**Lợi ích:**

* **Dễ mở rộng** (thêm hình mới chỉ cần tạo class mới, không sửa code cũ).
* **Giảm rủi ro bug do chỉnh sửa code hiện có**.

**5. Law of Demeter (Principle of Least Knowledge)**

* "Một module chỉ nên giao tiếp với các module **trực tiếp liên quan**, không nên truy cập sâu vào các thành phần con của đối tượng khác."

**Lợi ích của Law of Demeter:**

* Giảm sự phụ thuộc giữa các class.
* Dễ bảo trì vì thay đổi trong một class không ảnh hưởng sâu đến class khác.
* Hệ thống linh hoạt và dễ mở rộng hơn.

**Tóm tắt nhanh:**

* **Loose Coupling tốt hơn Tight Coupling** vì giúp code dễ bảo trì, mở rộng.
* **Composition tốt hơn Inheritance** vì linh hoạt, tránh ràng buộc chặt chẽ.
* **Immutable Objects** giúp thread-safe, giảm lỗi.
* **Open/Closed Principle** giúp mở rộng dễ dàng mà không sửa code cũ.
* **Law of Demeter** giảm phụ thuộc giữa các module, tăng tính linh hoạt.

**II. Practical OOP Exercises**

**1. Basic Exercises**

**1. Class Person với phương thức Introduce()**

**A computer screen shot of a program code

AI-generated content may be incorrect.**

**2. Class Shape và các class kế thừa Circle, Rectangle**

**A screenshot of a computer program

AI-generated content may be incorrect.** **A screenshot of a computer program

AI-generated content may be incorrect.** A screenshot of a computer program

AI-generated content may be incorrect.

**3. Class BankAccount với kiểm tra số dư không âm**

**A screenshot of a computer program

AI-generated content may be incorrect.** **A computer screen with text on it

AI-generated content may be incorrect.**

**2. Intermediate Exercises**

**1. Hierarchy Vehicle với phương thức Drive() sử dụng đa hình**

**A screen shot of a computer program

AI-generated content may be incorrect.** **A screenshot of a computer program

AI-generated content may be incorrect.** A computer screen shot of text

AI-generated content may be incorrect. A screen shot of a computer program

AI-generated content may be incorrect.

**2. Hệ thống thư viện quản lý sách, người mượn, và ngày trả**

**A computer screen shot of text

AI-generated content may be incorrect.** **A screen shot of a computer screen

AI-generated content may be incorrect.** A computer screen shot of a program

AI-generated content may be incorrect. A computer screen shot of a program

AI-generated content may be incorrect. A computer screen with white text

AI-generated content may be incorrect.

**3. Hệ thống thanh toán sử dụng Factory Pattern** **A black screen with white text

AI-generated content may be incorrect.** A screen shot of a computer

AI-generated content may be incorrect. A screen shot of a computer screen

AI-generated content may be incorrect. A screen shot of a computer program

AI-generated content may be incorrect. A computer code on a black background

AI-generated content may be incorrect.

**3. Advanced Exercises**

**1. Design an e-commerce application using OOP:**

**A computer screen shot of code

AI-generated content may be incorrect.** **A computer screen shot of a program

AI-generated content may be incorrect.** A screenshot of a computer program

AI-generated content may be incorrect. A screen shot of a computer program

AI-generated content may be incorrect.

**2. Hệ thống Plugin với Interface và Dynamic Loading**

**A screen shot of a computer program

AI-generated content may be incorrect.** **A computer screen shot of a program

AI-generated content may be incorrect.**

**3. Mô phỏng sở thú với đa hình và lớp trừu tượng**

**A screen shot of a computer program

AI-generated content may be incorrect.** **A screen shot of a computer program

AI-generated content may be incorrect.** A screen shot of a computer program

AI-generated content may be incorrect. A screen shot of a computer program

AI-generated content may be incorrect. A screen shot of a computer program

AI-generated content may be incorrect.