

BÀI 4. LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG

Nội dung

PHẦN 1. LỚP VÀ ĐỐI TƯỢNG.....	2
1. Tổng quan về lập trình hướng đối tượng.....	2
2. OOP là gì?	2
3. Lịch sử của OOP.....	3
4. Tại sao sử dụng OOP?.....	3
5. Các đặc điểm của OOP.....	5
5.1. Các đối tượng (Objects)	5
5.2. Đặc điểm 1: Tính trừu tượng (Abstraction).....	5
5.3. Đặc điểm 2: Tính đóng gói (Encapsulation)	5
5.4. Đặc điểm 3: Tính thừa kế (Inheritance)	6
5.5. Đặc điểm 4: Tính đa hình (Polymorphism).....	7
5.6. Tổng hợp (Aggregation).....	7
6. C # nhìn ở góc độ lịch sử.....	7
7. Tạo class	8
7.1. Giới thiệu các đối tượng và lớp (object và class).....	8
7.2. Định nghĩa lớp (class).....	8
7.3. Tạo các thuộc tính cho class	8
7.4. Tạo các phương thức	11
7.5. Phương thức khởi tạo	12
7.6. Nạp chồng (overloading).....	12
7.7. Luyện tập	13
7.8. Từ khóa static (tĩnh)	17
PHẦN 2. KẾ THỪA VÀ GIAO DIỆN	23
1. Kế thừa (Inheritance).....	23
1.1. Tạo class cơ sở và class dẫn xuất	23
1.2. Sử dụng quyền truy cập trong class.....	24
1.3. Ghi đè các phương thức của một class cơ sở	26
1.4. Tính đa hình trong lập trình hướng đối tượng.....	33
1.5. Lớp niêm phong (sealed class)	36
1.6. Lớp trừu tượng (class abstract).....	37
2. Giao diện (interface).....	38

2.1. Khai báo.....	39
2.2. Ví dụ tạo giao diện và thực thi	40
2.3. Giao diện IComparable	41
2.4. Giao diện IComparer	45

PHẦN 1. LỚP VÀ ĐỐI TƯỢNG

1. Tổng quan về lập trình hướng đối tượng

Để tạo tiền đề cho việc học lập trình hướng đối tượng (OOP) và C #, bài học này sẽ xem xét sơ qua về lịch sử của lập trình hướng đối tượng và các đặc điểm của ngôn ngữ lập trình hướng đối tượng.

Chúng ta sẽ xem tại sao lập trình hướng đối tượng lại trở nên quan trọng trong sự phát triển của các hệ thống phần mềm phân tán có sức mạnh công nghiệp. Chúng ta cũng sẽ xem cách C # đã phát triển thành một trong những ứng dụng hàng đầu ngôn ngữ lập trình.

Sau khi đọc bài học này, chúng ta sẽ nắm được:

- Lập trình hướng đối tượng là gì.
- Tại sao lập trình hướng đối tượng lại trở nên quan trọng trong việc phát triển các ứng dụng sức mạnh công nghiệp.
- Các đặc điểm tạo nên một ngôn ngữ lập trình hướng đối tượng.
- Lịch sử và sự phát triển của C #.

2. OOP là gì?

Lập trình hướng đối tượng (OOP) là một cách tiếp cận để phát triển phần mềm, trong đó cấu trúc của phần mềm được dựa trên các đối tượng tương tác với nhau để hoàn thành một nhiệm vụ. Sự tương tác này có dạng các thông điệp chuyển qua chuyển lại giữa các đối tượng. Để trả lời một thông điệp, một đối tượng có thể thực hiện một hành động.

Nếu nhìn vào cách chúng ta hoàn thành nhiệm vụ trong thế giới xung quanh, chúng ta có thể thấy rằng đúng là chúng ta đang tương tác trong một thế giới hướng đối tượng. Ví dụ, nếu muốn đến cửa hàng, bạn sẽ tương tác với một đối tượng ô tô. Một đối tượng ô tô bao gồm các các đối tượng tương tác với nhau để hoàn thành nhiệm vụ đưa bạn đến cửa hàng. Bạn đặt chìa khóa vào vật đánh lửa và vặn nó. Điều này đến lượt nó sẽ gửi một thông điệp (thông qua một tín hiệu điện) đến đối tượng khởi động, tương tác với đối tượng động cơ để khởi động xe. Là một người lái xe, bạn bị cô lập khỏi logic về cách các đối tượng của hệ thống hoạt động cùng nhau để khởi động xe. Bạn chỉ cần bắt đầu chuỗi sự kiện bằng cách thực hiện phương thức bắt đầu của đối tượng đánh lửa bằng chìa khóa. Sau đó, bạn chờ phản hồi (thông báo) thành công hay thất bại.

Tương tự như vậy, người dùng các chương trình phần mềm bị cô lập khỏi logic cần thiết để hoàn thành một nhiệm vụ. Ví dụ: khi bạn in một trang trong trình xử lý văn bản của mình, bạn bắt đầu hành động bằng cách nhấp vào nút in. Bạn chỉ cần đợi một phản hồi cho bạn biết nếu nó được in. Trong chương trình phần mềm, đối tượng nút tương tác với đối tượng máy in, đối tượng này tương tác với máy in thực tế để hoàn thành nhiệm vụ in trang.

3. Lịch sử của OOP

Các khái niệm OOP bắt đầu xuất hiện vào giữa những năm 1960 với ngôn ngữ lập trình gọi là Simula và ngày càng phát triển

vào những năm 1970 với sự ra đời của Smalltalk. Mặc dù các nhà phát triển phần mềm không áp dụng những tiến bộ ban đầu này trong ngôn ngữ OOP, nhưng các phương pháp luận hướng đối tượng vẫn tiếp tục phát triển. Vào giữa những năm 1980, mối quan tâm trở lại đối với các phương pháp luận hướng đối tượng. Cụ thể, các ngôn ngữ OOP như C++ và Eiffel đã trở nên phổ biến với các lập trình viên máy tính chính thống. OOP tiếp tục phát triển phổ biến trong những năm 1990, đáng chú ý nhất là với sự ra đời của Java và thu hút lượng lớn người theo dõi nó. Và vào năm 2002, cùng với việc phát hành .NET Framework, Microsoft đã giới thiệu một ngôn ngữ OOP mới, C# (phát âm là C-sharp) và cải tiến ngôn ngữ hiện có phổ biến rộng rãi của họ, Visual Basic, để bây giờ nó thực sự là hướng đối tượng. Ngày nay các ngôn ngữ OOP tiếp tục phát triển và là trụ cột của lập trình hiện đại.

4. Tại sao sử dụng OOP?

Tại sao OOP đã phát triển thành một mô hình được sử dụng rộng rãi như vậy để giải quyết các vấn đề nghiệp vụ ngày nay? Trong những năm 1970 và 1980, các ngôn ngữ lập trình hướng thủ tục như C, Pascal và Fortran đã được sử dụng rộng rãi để phát triển các hệ thống phần mềm theo định hướng nghiệp vụ. Các ngôn ngữ thủ tục tổ chức chương trình theo kiểu tuyến tính — chúng chạy từ trên xuống dưới. Nói cách khác, chương trình là một chuỗi các bước chạy nối tiếp nhau. Kiểu lập trình này hoạt động tốt đối với các chương trình nhỏ bao gồm vài trăm dòng mã, nhưng khi các chương trình trở nên lớn hơn, chúng trở nên khó quản lý và gỡ lỗi.

Trong nỗ lực quản lý kích thước ngày càng tăng của các chương trình, lập trình có cấu trúc đã được giới thiệu để chia nhỏ mã thành các phân đoạn có thể quản lý được gọi là các hàm hoặc thủ tục. Đây là một cải tiến, nhưng khi các chương trình thực hiện chức năng nghiệp vụ phức tạp hơn và tương tác với các hệ thống khác, những thiếu sót sau của lập trình cấu trúc bắt đầu xuất hiện:

- Các chương trình trở nên khó bảo trì hơn.
- Chức năng hiện có khó thay đổi mà không ảnh hưởng xấu đến tất cả chức năng của hệ thống.

- Các chương trình mới về cơ bản được xây dựng từ đầu. Do đó, có rất ít lợi nhuận từ việc đầu tư của những nỗ lực trước đó.
- Lập trình không có lợi cho việc phát triển nhóm. Các lập trình viên phải biết mọi khía cạnh về cách một chương trình hoạt động và không thể cô lập những nỗ lực của họ trên một khía cạnh của hệ thống.
- Thật khó để chuyển các mô hình nghiệp vụ thành mô hình lập trình.
- Lập trình cấu trúc hoạt động tốt khi tách biệt nhưng không tích hợp tốt với các hệ thống khác.

Ngoài những thiếu sót này, một số sự phát triển của hệ thống máy tính đã gây ra căng thẳng hơn nữa đối với cách tiếp cận chương trình cấu trúc, chẳng hạn như:

- Các nhà lập trình yêu cầu và có quyền truy cập trực tiếp vào các chương trình thông qua việc kết hợp các giao diện người dùng đồ họa và máy tính để bàn của họ.
- Người dùng yêu cầu một cách tiếp cận trực quan hơn, ít cấu trúc hơn để tương tác với các chương trình.
- Các hệ thống máy tính đã phát triển thành một mô hình phân tán trong đó logic nghiệp vụ, giao diện người dùng và cơ sở dữ liệu phụ trợ được kết hợp lỏng lẻo và được truy cập qua Internet và mạng nội bộ.

Do đó, nhiều nhà phát triển phần mềm nghiệp vụ đã chuyển sang các phương pháp hướng đối tượng và ngôn ngữ lập trình.

Ngôn ngữ lập trình dựa trên hai khái niệm cơ bản là dữ liệu và cách thức để thao tác với dữ liệu.

Cách tiếp cận hướng đối tượng định nghĩa các đối tượng là các thực thể có một tập giá trị xác định và một tập hợp các thao tác xác định có thể được thực hiện trên các giá trị này.

Những lợi ích bao gồm những điều sau đây:

- Chuyển đổi trực quan hơn từ mô hình phân tích kinh doanh sang mô hình triển khai phần mềm.
- Khả năng duy trì và thực hiện các thay đổi trong chương trình một cách hiệu quả và nhanh chóng hơn.
- Khả năng tạo hệ thống phần mềm hiệu quả hơn bằng cách sử dụng quy trình nhóm, cho phép các chuyên gia làm việc trên các bộ phận của hệ thống.
- Khả năng sử dụng lại các thành phần mã trong các chương trình khác và mua các thành phần được viết bởi các nhà phát triển bên thứ ba để tăng chức năng của các chương trình hiện có mà không tốn nhiều công sức.
- Tích hợp tốt hơn với các hệ thống máy tính phân tán được kết hợp lỏng lẻo.
- Cải tiến tích hợp với các hệ điều hành hiện đại.
- Khả năng tạo giao diện đồ họa-người dùng trực quan hơn cho người dùng.

5. Các đặc điểm của OOP

Trong phần này, bạn sẽ xem xét một số khái niệm và thuật ngữ cơ bản chung cho tất cả các ngôn ngữ OOP.

5.1. Các đối tượng (Objects)

Như đã lưu ý trước đó, chúng ta đang sống trong một thế giới hướng đối tượng. Bạn là một đối tượng. Bạn tương tác với các đối tượng khác. Trên thực tế, bạn là một đối tượng có dữ liệu như chiều cao và màu tóc của bạn. Bạn cũng có các hành vi mà bạn thực hiện hoặc được thực hiện trên bạn, chẳng hạn như ăn uống và đi bộ.

Vậy đối tượng là gì? Theo thuật ngữ OOP, một đối tượng là một cấu trúc kết hợp dữ liệu và các cách thức để làm việc với dữ liệu đó. Ví dụ: nếu bạn muốn có chức năng in trong ứng dụng của mình, bạn sẽ làm việc với một đối tượng máy in chịu trách nhiệm về dữ liệu và phương pháp được sử dụng để tương tác với máy in của bạn.

5.2. Đặc điểm 1: Tính trừu tượng (Abstraction)

Khi bạn tương tác với các đối tượng trên thế giới, bạn thường chỉ quan tâm đến một tập hợp con các thuộc tính của chúng. Nếu không có khả năng này để trừu tượng hóa hoặc lọc ra các thuộc tính không liên quan của các đối tượng, bạn sẽ khó xử lý vô số thông tin đang làm bạn nhiễu loạn và khó tập trung vào nhiệm vụ trước mắt.

Kết quả của sự trừu tượng hóa, khi hai người khác nhau tương tác với cùng một đối tượng, họ thường xử lý một tập hợp con các thuộc tính khác nhau. Ví dụ, khi tôi lái xe ô tô của mình, tôi cần biết tốc độ của chiếc xe và hướng đi của nó.

Vì xe sử dụng hộp số tự động nên tôi không cần biết số vòng quay trên phút (RPM) của động cơ nên tôi lọc thông tin này ra. Mặt khác, thông tin này sẽ rất quan trọng đối với một tay đua xe đua, người sẽ không lọc thông tin này ra.

Khi xây dựng các đối tượng trong các ứng dụng OOP, điều quan trọng là phải kết hợp khái niệm trừu tượng này. Các đối tượng chỉ bao gồm thông tin có liên quan trong ngữ cảnh của ứng dụng. Nếu bạn đang xây dựng một ứng dụng vận chuyển, bạn sẽ tạo một đối tượng sản phẩm với các thuộc tính như kích thước và trọng lượng. Màu sắc của mặt hàng sẽ là thông tin không liên quan và sẽ bị bỏ qua. Mặt khác, khi xây dựng ứng dụng nhập đơn đặt hàng, màu sắc có thể quan trọng và sẽ được đưa vào như một thuộc tính của đối tượng sản phẩm.

5.3. Đặc điểm 2: Tính đóng gói (Encapsulation)

Một tính năng quan trọng khác của OOP là tính đóng gói. Đóng gói là quá trình mà không có quyền truy cập trực tiếp nào được phép cấp vào dữ liệu; thay vào đó, nó bị ẩn. Nếu bạn muốn có quyền truy cập vào dữ liệu, bạn phải tương tác với đối tượng chịu trách nhiệm về dữ liệu. Trong ví dụ kiểm kê trước đó, nếu bạn muốn xem hoặc cập nhật thông tin về sản phẩm, bạn sẽ phải làm việc thông qua đối tượng sản phẩm. Để đọc dữ liệu, bạn sẽ gửi cho đối tượng sản phẩm một thông điệp. Các đối tượng sản phẩm sau

đó sẽ đọc giá trị và gửi lại một thông điệp cho bạn biết giá trị là gì. Đối tượng sản phẩm xác định những thao tác nào có thể được thực hiện trên dữ liệu sản phẩm. Nếu bạn gửi một thông điệp để sửa đổi dữ liệu và đối tượng sản phẩm xác định đó là một yêu cầu hợp lệ, nó sẽ thực hiện thao tác cho bạn và gửi lại một thông điệp kèm theo kết quả.

Bạn trải nghiệm sự gói gọn trong cuộc sống hàng ngày của mình mọi lúc. Hãy nghĩ về một bộ phận nhân sự. Họ đóng gói (ẩn) thông tin về nhân viên. Họ xác định cách dữ liệu này có thể được sử dụng và thao tác. Bất kỳ yêu cầu nào đối với dữ liệu nhân viên hoặc yêu cầu cập nhật dữ liệu đều phải được chuyển qua chúng. Một ví dụ khác là an ninh mạng. Mọi yêu cầu về thông tin bảo mật hoặc thay đổi chính sách bảo mật phải được thực hiện thông qua quản trị viên an ninh mạng. Dữ liệu bảo mật được đóng gói từ những người sử dụng mạng.

Bằng cách đóng gói dữ liệu, bạn làm cho dữ liệu của hệ thống của mình an toàn và đáng tin cậy hơn. Bạn biết dữ liệu đang được truy cập như thế nào và những thao tác nào đang được thực hiện trên dữ liệu. Điều này làm cho việc bảo trì chương trình dễ dàng hơn nhiều và cũng đơn giản hóa quá trình gỡ lỗi. Bạn cũng có thể sửa đổi các phương thức được sử dụng để làm việc trên dữ liệu và, nếu bạn không thay đổi cách phương thức được yêu cầu và kiểu phản hồi được gửi lại, bạn không cần phải thay đổi các đối tượng khác bằng phương pháp này. Hãy suy nghĩ về thời điểm bạn gửi một bức thư qua đường bưu điện. Bạn yêu cầu bưu điện chuyển thư. Làm thế nào bưu điện hoàn thành việc này không được tiết lộ cho bạn. Nếu nó thay đổi lộ trình mà nó sử dụng để gửi thư, nó không ảnh hưởng đến cách bạn bắt đầu gửi thư. Bạn không cần phải biết các thủ tục nội bộ của bưu điện được sử dụng để chuyển thư.

5.4. Đặc điểm 3: Tính thừa kế (Inheritance)

Hầu hết các đối tượng trong cuộc sống thực có thể được phân loại thành các thứ bậc. Ví dụ, bạn có thể phân loại tất cả các con chó với nhau vì có một số đặc điểm chung nhất định như có bốn chân và lông. Các giống của họ tiếp tục phân loại chúng thành các nhóm phụ với các thuộc tính chung như kích thước và phong thái. Bạn cũng phân loại các đối tượng theo chức năng của chúng. Ví dụ, có xe thương mại và xe giải trí. Có xe tải và xe khách. Bạn phân loại ô tô theo thực hiện và mô hình của họ. Để hiểu thế giới, bạn cần sử dụng phân cấp và phân loại đối tượng.

Bạn sử dụng tính kế thừa trong OOP để phân loại các đối tượng trong chương trình của mình theo các đặc điểm và chức năng chung. Điều này giúp làm việc với các đối tượng dễ dàng và trực quan hơn. Nó cũng làm cho việc lập trình dễ dàng hơn vì nó cho phép bạn kết hợp các đặc điểm chung vào một đối tượng mẹ và kế thừa những đặc điểm này trong các đối tượng con. Ví dụ: bạn có thể xác định một đối tượng nhân viên xác định tất cả các đặc điểm chung của nhân viên trong công ty của bạn. Sau đó, bạn có thể xác định đối tượng người quản lý kế thừa các đặc điểm của đối tượng nhân viên nhưng cũng bổ sung các đặc điểm riêng cho người quản lý trong công ty của bạn. Do tính kế thừa,

đối tượng quản lý sẽ tự động phản ánh bất kỳ thay đổi nào đối với các đặc điểm của đối tượng nhân viên.

5.5. Đặc điểm 4: Tính đa hình (Polymorphism)

Tính đa hình là khả năng các đối tượng khác nhau phản hồi cùng một thông điệp yêu cầu theo cách riêng của chúng.

Điều này liên quan như thế nào đến OOP? Bạn có thể tạo các đối tượng phản hồi cùng một thông báo trong các triển khai độc đáo của riêng chúng. Ví dụ: bạn có thể gửi tin nhắn đến một đối tượng máy in sẽ in văn bản trên máy in và bạn có thể gửi thông báo tương tự đến một đối tượng màn hình sẽ in văn bản đến một cửa sổ trên màn hình máy tính của bạn.

Một ví dụ điển hình khác về tính đa hình là việc sử dụng các từ trong ngôn ngữ tiếng Anh. Các từ có nhiều nghĩa khác nhau, nhưng thông qua ngữ cảnh của câu, bạn có thể suy ra ý nghĩa nào được sử dụng. Bạn biết rằng ai đó nói “Give me a break!” (Hãy cho tôi nghỉ ngơi!) không phải là yêu cầu bạn đánh gãy chân của mình!

5.6. Tổng hợp (Aggregation)

Tổng hợp là khi một đối tượng bao gồm một tổng hợp các đối tượng khác hoạt động cùng nhau. Ví dụ: đối tượng máy cắt cỏ của bạn là tổng hợp của các đối tượng bánh xe, đối tượng động cơ, đối tượng lưỡi dao, v.v. Trên thực tế, đối tượng động cơ là tổng hợp của nhiều đối tượng khác. Có rất nhiều ví dụ về tập hợp trên thế giới xung quanh chúng ta. Khả năng sử dụng tính năng tổng hợp trong OOP là một tính năng mạnh mẽ cho phép bạn lập mô hình và thực hiện chính xác các quy trình nghiệp vụ trong chương trình của mình.

Bây giờ bạn đã hiểu về những gì cấu thành ngôn ngữ OOP và tại sao ngôn ngữ OOP lại quan trọng đối với việc phát triển ứng dụng cấp doanh nghiệp, bước tiếp theo của bạn là làm quen với cách các ứng dụng OOP được thiết kế.

6. C # nhìn ở góc độ lịch sử

Ngôn ngữ C # được lấy cảm hứng từ Java. Đến lượt nó, Java được lấy cảm hứng từ C ++, một lần nữa - về mặt hướng đối tượng - có thể được truy ngược trở lại Simula.

Dưới đây là tổng quan về các ngôn ngữ lập trình hướng đối tượng quan trọng nhất mà từ đó C # đã được hình thành:

- Simula (1967) - Ngôn ngữ lập trình hướng đối tượng đầu tiên
- C ++ (1983) - Ngôn ngữ lập trình hướng đối tượng đầu tiên trong họ ngôn ngữ C
- Java (1995) - Ngôn ngữ lập trình hướng đối tượng của Sun
- C # (2001) - Ngôn ngữ lập trình hướng đối tượng của Microsoft

7. Tạo class

7.1. Giới thiệu các đối tượng và lớp (object và class)

Trong OOP, bạn sử dụng các đối tượng trong chương trình của mình để đóng gói dữ liệu được liên kết với các thực thể mà chương trình đang hoạt động. Ví dụ, một ứng dụng nhân sự cần làm việc với nhân viên. Nhân viên có các thuộc tính liên quan đến họ cần được theo dõi. Bạn có thể quan tâm đến những thứ như tên nhân viên, địa chỉ, phòng ban, v.v. Mặc dù bạn theo dõi các thuộc tính giống nhau cho tất cả nhân viên, nhưng mỗi nhân viên có các giá trị riêng cho các thuộc tính này. Trong ứng dụng nguồn nhân lực, một đối tượng Nhân viên lấy và sửa đổi các thuộc tính được liên kết với một nhân viên.

Trong OOP, các đặc tính (attribute) của một đối tượng được gọi là thuộc tính (property). Cùng với các thuộc tính của nhân viên, ứng dụng nhân sự cũng cần một tập hợp các hành vi được thiết lập bởi đối tượng Nhân viên. Ví dụ, một hành vi quan tâm của nhân viên đối với bộ phận nhân sự là khả năng yêu cầu thời gian nghỉ. Trong OOP, các đối tượng thể hiện các hành vi thông qua các phương thức. Đối tượng Employee chứa phương thức RequestTimeOff chứa mã thực thi.

Các thuộc tính và phương thức của các đối tượng được sử dụng trong OOP được xác định thông qua các lớp (class). Class là một bản thiết kế định nghĩa các thuộc tính và hành vi của các đối tượng được tạo ra như các thể hiện của class. Nếu bạn đã hoàn thành việc phân tích và thiết kế ứng dụng phù hợp, bạn có thể tham khảo tài liệu thiết kế UML để xác định những class nào cần được xây dựng và những thuộc tính và phương thức mà các class này sẽ chứa. Các biểu đồ lớp UML chứa thông tin ban đầu bạn cần để xây dựng các class của hệ thống.

Chúng ta sẽ xem mã của một lớp Nhân viên (Employee) đơn giản. Lớp Employee sẽ có các thuộc tính và phương thức và làm việc với dữ liệu nhân viên như một phần của ứng dụng nhân lực giả định.

7.2. Định nghĩa lớp (class)

Chúng ta hãy kiểm tra mã nguồn cần thiết để tạo định nghĩa class. Dòng mã đầu tiên xác định khối mã là định nghĩa class bằng cách sử dụng lớp từ khóa theo sau là tên của class. Phần thân của định nghĩa class được bao gồm bởi một dấu ngoặc nhọn mở và đóng.

Khối mã được cấu trúc như sau:

```
class Person
{
}
```

7.3. Tạo các thuộc tính cho class

Sau khi xác định điểm bắt đầu { và điểm kết thúc của khối mã } class, bước tiếp theo là xác định các biến thể hiện (thường được gọi là trường) có trong class. Các biến này giữ

dữ liệu mà một thể hiện của class của bạn sẽ thao tác. Từ khóa private đảm bảo rằng các biến cá thể này chỉ có thể được thao tác bởi mã bên trong class. Dưới đây là các định nghĩa về biến thể hiện:

```
private string _id;
private string _name;
private int _age;
```

Khi người dùng của class (máy khách) cần truy vấn hoặc đặt giá trị của các biến thể hiện này, các thuộc tính công khai sẽ hiển thị với chúng. Bên trong khối mã thuộc tính là khối Set (nhận) và khối Get (đặt). Khối Get trả về giá trị của biến riêng tư cho người dùng của class. Mã này cung cấp một thuộc tính chỉ có thể đọc. Khối Set cung cấp một thuộc tính cho phép ghi; nó chuyển một giá trị do người dùng gửi đến biến thể hiện riêng tương ứng. Đây là một ví dụ về khối thuộc tính:

```
public string Id
{
    get { return _id; }
    set { _id = value; }
}

public string Name
{
    get { return _name; }
    set { _name = value; }
}

public int Age
{
    get { return _age; }
    set { _age = value; }
}
```

Có thể đôi khi bạn muốn hạn chế quyền truy cập vào một thuộc tính để người dùng có thể đọc giá trị thuộc tính nhưng không thay đổi nó. Bằng cách loại bỏ khối thiết lập bên trong khối thuộc tính, bạn tạo thuộc tính chỉ đọc. Đoạn mã sau cho biết cách đặt thuộc tính EmployeeID ở chế độ chỉ đọc:

```
public string Id
{
    get { return _id; }
}
```

Những người mới dùng OOP thường hỏi tại sao cần phải trải qua quá nhiều công việc để có và thiết lập thuộc tính. Bạn không thể chỉ tạo các biến phiên bản công khai mà người dùng có thể đọc và ghi trực tiếp? Câu trả lời nằm ở một trong những nguyên lý cơ bản của OOP: tính đóng gói. Đóng gói có nghĩa là máy khách không có quyền truy cập trực tiếp vào dữ liệu. Khi làm việc với dữ liệu, máy khách phải sử dụng các thuộc tính và phương thức được xác định rõ ràng được truy cập thông qua một thể hiện của lớp.

```
static void Main(string[] args)
```

```
{
    Person p = new Person();
    p.Id = "P01";
    p.Name = "Nguyen Thi Anh";
    p.Age = 20;
    Console.WriteLine("Person Id: " + p.Id);
    Console.WriteLine("Person Name: " + p.Name);
    Console.WriteLine("Person Age: " + p.Age);
}
```

Như vậy Id, Name, Age là các phương thức được dùng để thiết lập hoặc lấy ra (Set/Get) các giá trị của các thuộc tính của class.

Sau đây là một số lợi ích của việc đóng gói dữ liệu theo cách này:

- Ngăn chặn truy cập trái phép vào dữ liệu
- Đảm bảo tính toàn vẹn của dữ liệu thông qua kiểm tra lỗi
- Tạo thuộc tính chỉ đọc hoặc chỉ ghi
- Cách ly người dùng của lớp khỏi những thay đổi trong mã triển khai. Ví dụ: bạn có thể kiểm tra để đảm bảo mật khẩu dài ít nhất sáu ký tự thông qua đoạn mã sau:

```
public string Password
{
    get { return _password; }
    set
    {
        if (value.Length >= 6)
        {
            _password = value;
        }
        else
        {
            throw new Exception("Password must be at least 6 characters");
        }
    }
}
```

*Thuộc tính được thực thi tự động

Trong một số trường hợp, người truy cập thuộc tính get và set chỉ cần gán một giá trị cho hoặc truy xuất một giá trị từ trường hỗ trợ mà không bao gồm bất kỳ logic bổ sung nào. Bằng cách sử dụng các thuộc tính được triển khai tự động, bạn có thể đơn giản hóa mã của mình trong khi trình biên dịch C # cung cấp trường hỗ trợ cho bạn một cách minh bạch.

Nếu một thuộc tính có cả trình truy cập get và set (hoặc get và init), thì cả hai đều phải được triển khai tự động. Bạn xác định thuộc tính được triển khai tự động bằng cách sử dụng các từ khóa get và set mà không cần cung cấp bất kỳ triển khai nào.

Với class Person của chúng ta ở trên, đoạn mã trở nên ngắn gọn như sau:

```
class Person
{
    public string Id{ get; set; }
    public string Name{ get; set; }
    public int Age{ get; set; }
}
```

Và khi tạo các thể hiện (đối tượng):

```
Person p = new Person();
p.Id = "P01";
p.Name = "Nguyễn Thị Anh";
p.Age = 20;
```

***Lưu ý:**

Trong ví dụ trên, các thuộc tính đang được đặt ký tự đầu tiên là chữ hoa. Để thuận lợi cho quá trình viết mã, sau này chúng ta sẽ đặt tên cho các thuộc tính là chữ thường.

7.4. Tạo các phương thức

Các phương thức của class xác định các hành vi của class. Ví dụ: đoạn mã sau xác định một phương thức cho class Person, xác minh việc người này có đủ tuổi thành niên hay chưa:

```
public bool CheckAge()
{
    if (age >= 18)
    {
        return true;
    }
    return false;
}
```

Xét một ví dụ khác, chúng ta có class hình chữ nhật (rectangle) có 2 thuộc tính là chiều dài (height) và chiều rộng (width). Ngoài việc sử dụng thuộc tính thực thi tự động, chúng ta sẽ viết thêm 2 phương thức là tính diện tích (area) và chu vi (perimeter) cho hình chữ nhật. Đoạn mã như sau:

```
class Rectangle
{
    public int height{ get; set; }
    public int width { get; set; }
    public int Area()
    {
        return height * width;
    }

    public int Perimeter()
    {
        return (height + width) * 2;
    }
}
```

Trong phương thức main:

```
Rectangle r = new Rectangle();
r.height = 8;
r.width = 9;
Console.WriteLine("Area is: "+r.Area());
Console.WriteLine("Perimeter is: "+r.Perimeter());
```

Kết quả thực hiện:

Microsoft Visual Studio Debug Console

Area is: 72

Perimeter is: 34

7.5. Phương thức khởi tạo

Mỗi khi nào một class được tạo, phương thức khởi tạo của class sẽ được tự động gọi. Một class có thể có nhiều phương thức khởi tạo với các tham số khác nhau. Phương thức khởi tạo cho phép người lập trình thiết lập các giá trị mặc định, hoặc giới hạn việc khởi tạo và viết mã linh hoạt và dễ đọc.

7.5.1. Phương thức khởi tạo không tham số

Nếu bạn không cung cấp một phương thức khởi tạo cho class của mình, C# sẽ tạo một phương thức khởi tạo mặc định để khởi tạo đối tượng và đặt các thuộc tính (biến thành viên) là các giá trị mặc định được quy định trong ngôn ngữ C#.

7.5.2. Cú pháp của phương thức khởi tạo

Phương thức có tên giống với tên class. Nguyên mẫu của phương thức kiểu này chỉ bao gồm tên phương thức và danh sách tham số của nó; không có kiểu dữ liệu trả về. Ví dụ sau đây cho thấy phương thức khởi tạo cho một lớp có tên là Person.

```
public Person(string id, string name, int age)
{
    this.id = id;
    this.name = name;
    this.age = age;
}
```

Tên các thuộc tính của class Person là id, name, age và tham số truyền vào phương thức khởi tạo ở trên đặt trùng tên với nhau, để phân biệt, chúng ta sử dụng từ khóa **this** ở bên trái phép gán để nói rằng đây là các thuộc tính của class và bên phải là các biến của phương thức.

Lưu ý: đặt tên tham số trong phương thức khởi tạo là tùy ý. Trường hợp không đặt trùng với tên thuộc tính thì chúng ta không cần thêm từ khóa **this**.

```
public Person(string id, string personname, int age)
{
    this.id = id;
    name = personname;
    this.age = age;
}
```

7.6. Nạp chồng (overloading)

Khả năng **nạp chồng** các phương thức là một tính năng hữu ích của các ngôn ngữ OOP. Bạn nạp chồng các phương thức trong một class bằng cách xác định nhiều phương thức có cùng tên nhưng chứa các chữ ký (nguyên mẫu) khác nhau. Chữ ký phương thức là sự kết hợp giữa tên của phương thức và danh sách kiểu tham số của nó. Nếu bạn thay đổi

danh sách kiểu tham số, bạn sẽ tạo một chữ ký phương thức khác. Ví dụ, danh sách kiểu tham số có thể chứa một số lượng tham số khác nhau hoặc các kiểu tham số khác nhau. Trình biên dịch sẽ xác định phương thức nào sẽ thực thi bằng cách kiểm tra danh sách kiểu tham số được người dùng chuyển vào.

```
public Person()
{
    id = "DEFAULT_ID";
    name = "DEFAULT_NAME";
    age = 0;
}
public Person(string id)
{
    this.id = id;
}
public Person(string id, string name, int age)
{
    this.id = id;
    this.name = name;
    this.age = age;
}
```

Trong ví dụ trên, có ba phương thức khởi tạo. Phương thức thứ nhất không có tham số, phương thức thứ hai có một tham số và phương thức thứ ba có tất cả các tham số ứng với số thuộc tính của class.

7.7. Luyện tập

Luyện tập 1

Tạo một class để lưu trữ thông tin chi tiết của sinh viên như rollno, name, course joined and fee paid (mã đăng ký, tên, khóa học đã tham gia và học phí đã trả). Giả sử các khóa học là C# và ASP.NET với học phí là 2000 và 3000.

Phương thức khởi tạo có ba tham số rollno, tên và khóa học.

Chúng ta có các thuộc tính và phương thức sau:

- ✓ Thuộc tính DueAmount
- ✓ Thuộc tính TotalFee
- ✓ Phương thức Payment(amount)
- ✓ Phương thức Print()

```
class Student
{
    private int rollno;
    private string name;
    private string course;
    private int feepaid;

    public Student(int rollno, string name, string course)
    {
        this.rollno = rollno;
        this.name = name;
        this.course = course;
    }
}
```

```

    }

    public void Payment(int amount)
    {
        feepaid += amount;
    }

    public void Print()
    {
        Console.WriteLine("RollNo: "+rollno);
        Console.WriteLine("Name: "+name);
        Console.WriteLine("Course:"+course);
        Console.WriteLine("FeePaid: "+feepaid);
    }

    public int DueAmount
    {
        get
        {
            return TotalFee - feepaid;
        }
    }

    public int TotalFee
    {
        get
        {
            return course == "C#" ? 2000 : 3000;
        }
    }
}

class UseStudent
{
    public static void Main()
    {
        Console.WriteLine("==== s1 =====");
        Student s1 = new Student(1, "Anna", "C#");
        s1.Payment(800);
        s1.Print();
        Console.WriteLine(s1.DueAmount);
        Console.WriteLine("==== s2 =====");
        Student s2 = new Student(2, "John", "ASP.NET");
        s2.Payment(1600);
        s2.Print();
        Console.WriteLine(s2.DueAmount);
    }
}
}

```

Kết quả thực hiện:


```

C:\ Microsoft Visual Studio Debug Console

===== s1 =====
RollNo: 1
Name: Anna
Course:C#
FeePaid: 800
1200
===== s2 =====
RollNo: 2
Name: John
Course:ASP.NET
FeePaid: 1600
1400

```

Luyện tập 2

Tạo class Employee có các thuộc tính: id, name, age, salary, commision và total (định danh, tên, tuổi, lương, thưởng và tổng tiền).

Viết các phương thức khởi tạo không tham số, một tham số, năm tham số (trừ tham số total).

Thiết lập giá trị cho thuộc tính total bằng cách cộng salary và commision.

Viết phương thức Input() để nhập dữ liệu từ bàn phím cho các thuộc tính.

Viết phương thức Output() để hiển thị ra màn hình các thông tin của Employee.

Mã của chương trình như sau:

```

class Employee
{
    public string id { get; set; }
    public string name { get; set; }
    public int age { get; set; }
    public int salary { get; set; }
    public int commision { get; set; }

    public int total
    {
        get
        {
            return salary + commision; ;
        }
    }

    //Phương thức khởi tạo 0 tham số
    public Employee()
    {
        id = "Default_id";
        name = "Default_name";
        age = 0;
        salary = 0;
        commision = 0;
    }
}

```

```
//Phương thức khởi tạo 5 tham số
public Employee(string id, string name, int age, int salary, int comm)
{
    this.id = id;
    this.name = name;
    this.age = age;
    this.salary = salary;
    commision = comm;
}

//Phương thức khởi tạo 1 tham số
public Employee(string id)
{
    this.id = id;
}

public void Input()
{
    Console.WriteLine("id:");
    id = Console.ReadLine();
    Console.WriteLine("name:");
    name = Console.ReadLine();
    Console.WriteLine("age:");
    age = Convert.ToInt32(Console.ReadLine());
    Console.WriteLine("salary:");
    salary = Convert.ToInt32(Console.ReadLine());
    Console.WriteLine("commision:");
    commision = Convert.ToInt32(Console.ReadLine());
}

public void Output()
{
    Console.WriteLine("===== Employee information =====");
    Console.WriteLine("id: " + id);
    Console.WriteLine("name: " + name);
    Console.WriteLine("age: " + age);
    Console.WriteLine("salary: " + salary);
    Console.WriteLine("commision: " + commision);
    Console.WriteLine("total: " + total);
}
}
```

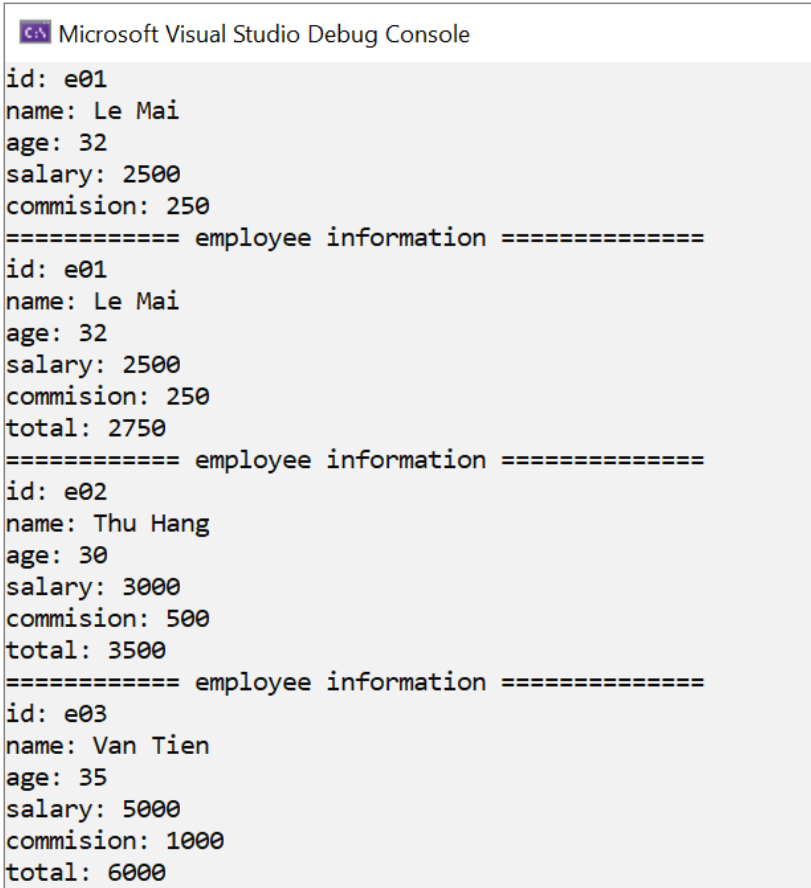
Mã trong phương thức Main()

```
//Sử dụng phương thức khởi tạo 0 tham số
Employee emp1 = new Employee();
emp1.Input();
emp1.Output();

//Sử dụng phương thức khởi tạo 5 tham số
Employee emp2 = new Employee("e02", "Thu Hang", 30, 3000, 500);
emp2.Output();
```

```
//Sử dụng phương thức khởi tạo 1 tham số
Employee emp3 = new Employee("e03");
emp3.name = "Van Tien";
emp3.age = 35;
emp3.salary = 5000;
emp3.commission = 1000;
emp3.Output();
```

Kết quả thực hiện:



```
Microsoft Visual Studio Debug Console
id: e01
name: Le Mai
age: 32
salary: 2500
commision: 250
===== employee information =====
id: e01
name: Le Mai
age: 32
salary: 2500
commision: 250
total: 2750
===== employee information =====
id: e02
name: Thu Hang
age: 30
salary: 3000
commision: 500
total: 3500
===== employee information =====
id: e03
name: Van Tien
age: 35
salary: 5000
commision: 1000
total: 6000
```

7.8. Từ khóa static (tĩnh)

7.8.1. Lớp tĩnh (static class)

Một static class về cơ bản giống như một class non-static, nhưng có một điểm khác biệt: một static class không thể được khởi tạo. Nói cách khác, bạn không thể sử dụng toán tử **new** để tạo một biến kiểu class.

Bởi vì không có biến thể hiện, chúng ta truy cập các thành viên của một class static bằng cách sử dụng chính tên class đó. Ví dụ: nếu bạn có một static class được đặt tên là **UtilityClass**, và có một phương thức tĩnh công khai có tên **MethodA**, bạn gọi phương thức như được hiển thị trong ví dụ sau:

```
UtilityClass.MethodA();
```

Một static class có thể được sử dụng như một vùng chứa thuận tiện cho các tập hợp các phương thức chỉ hoạt động trên các tham số truyền vào và không phải lấy hoặc đặt bất kỳ trường cá thể nội bộ nào. Ví dụ, trong Thư viện lớp .NET, lớp System.Math tĩnh chứa các phương thức thực hiện các phép toán học mà không có bất kỳ yêu cầu nào để lưu trữ hoặc truy xuất dữ liệu duy nhất cho một phiên bản cụ thể của lớp Math. Tức là, bạn áp dụng các thành viên của lớp bằng cách chỉ định tên lớp và tên phương thức, như được hiển thị trong ví dụ sau:

```
double dub = -3.14;
Console.WriteLine(Math.Abs(dub));
Console.WriteLine(Math.Floor(dub));
Console.WriteLine(Math.Round(Math.Abs(dub)));
```

Hiển thị:

```
// 3.14
// -4
// 3
```

Danh sách sau cung cấp các tính năng chính của một static class:

- ✓ Chỉ chứa các thành viên static.
- ✓ Không thể được khởi tạo.
- ✓ Bị niêm phong (sealed).
- ✓ Không thể chứa các thể hiện của phương thức khởi tạo.

Do đó, việc tạo một static class về cơ bản cũng giống như việc tạo một class chỉ chứa các thành viên static và một phương thức khởi tạo riêng. Một phương thức khởi tạo là private ngăn không cho lớp được khởi tạo. Ưu điểm của việc sử dụng một static class là trình biên dịch có thể kiểm tra để đảm bảo rằng không có thành viên cá thể nào được thêm vào một cách tình cờ. Trình biên dịch sẽ đảm bảo rằng các thể hiện của lớp này không thể được tạo.

Các static class được niêm phong và do đó không thể được kế thừa. Chúng không thể kế thừa từ bất kỳ lớp nào ngoại trừ đối tượng. Các static class không thể chứa một phương thức khởi tạo thể hiện. Tuy nhiên, chúng có thể chứa một phương thức khởi tạo static.

Ví dụ 1: Tạo static class và có hai phương thức.

```
static class TemperatureConverter
{
    public static double CelsiusToFahrenheit(string temperatureCelsius)
    {
        // Chuyển đổi chuỗi ký tự thành số để tính toán.
        double celsius = Double.Parse(temperatureCelsius);

        // Chuyển từ nhiệt độ Celsius sang nhiệt độ Fahrenheit.
        double fahrenheit = (celsius * 9 / 5) + 32;

        return fahrenheit;
    }
}
```

```

    }

    public static double FahrenheitToCelsius(string temperatureFahrenheit)
    {
        // Chuyển đổi chuỗi ký tự thành số để tính toán.
        double fahrenheit = Double.Parse(temperatureFahrenheit);

        // huyển từ nhiệt độ Fahrenheit sang nhiệt Celsius.
        double celsius = (fahrenheit - 32) * 5 / 9;

        return celsius;
    }
}

```

Mã trong phương thức Main():

```

static void Main(string[] args)
{
    string selection = "";
    do
    {
        Console.WriteLine("\nPlease select the convertor direction");
        Console.WriteLine("1. From Celsius to Fahrenheit.");
        Console.WriteLine("2. From Fahrenheit to Celsius.");
        Console.WriteLine("3. Finish.");

        Console.Write("Select:");

        selection = Console.ReadLine();
        double F, C = 0;

        switch (selection)
        {
            case "1":
                Console.Write("Please enter the Celsius temperature: ");
                F =
                TemperatureConverter.CelsiusToFahrenheit(Console.ReadLine());
                Console.WriteLine("Temperature in Fahrenheit: {0:F2}", F);
                break;

            case "2":
                Console.Write("Please enter the Fahrenheit temperature: ");
                C =
                TemperatureConverter.FahrenheitToCelsius(Console.ReadLine());
                Console.WriteLine("Temperature in Celsius: {0:F2}", C);
                break;

            default:
                Console.WriteLine("Please select a convertor.");
                break;
        }
    } while (!selection.Equals("3"));
}

```

Kết quả thực hiện:

```
E:\Camtasia-video-made\Lap trinh .net\000 ver2\Proj_oop_demo\

Please select the convertor direction
1. From Celsius to Fahrenheit.
2. From Fahrenheit to Celsius.
3. Finish.
Select:1
Please enter the Celsius temperature: 38
Temperature in Fahrenheit: 100.40

Please select the convertor direction
1. From Celsius to Fahrenheit.
2. From Fahrenheit to Celsius.
3. Finish.
Select:2
Please enter the Fahrenheit temperature: 100
Temperature in Celsius: 37.78

Please select the convertor direction
1. From Celsius to Fahrenheit.
2. From Fahrenheit to Celsius.
3. Finish.
Select:
```

Ví dụ 2: Tạo static class và có phương thức khởi tạo.

```
public static class NumberDemo
{
    public static int x;
    public static int y;

    //không cho phép là private, public, không cho phép có tham số.
    static NumberDemo()
    {
        x = 10;
        y = 5;
    }
    public static int Add()
    {
        return x + y;
    }
}
```

Trong main(), đoạn mã như sau:

```
Console.WriteLine("rs: " + NumberDemo.Add()); //15
NumberDemo.x = 100;
NumberDemo.y = 200;
Console.WriteLine("rs: " + NumberDemo.Add()); //300
```

7.8.2. Các thành viên trong lớp là tĩnh

Trong một class thông thường, có thể có các thành viên là thuộc tính hoặc phương thức được khai báo là static.

Phương thức là static chỉ có thể làm việc với các biến là static.

```
public class Employee4
{
    public string id;
    public string name;
```



```

public Employee4()
{
}

public Employee4(string name, string id)
{
    this.name = name;
    this.id = id;
}

public static int employeeCounter;

public static int AddEmployee()
{
    return ++employeeCounter;
}
}

class MainClass : Employee4
{
    static void Main()
    {
        Console.Write("Enter the employee's name: ");
        string name = Console.ReadLine();
        Console.Write("Enter the employee's ID: ");
        string id = Console.ReadLine();

        // Tạo và cấu hình đối tượng employee e.
        Employee4 e = new Employee4(name, id);
        Console.Write("Enter the current number of employees: ");
        string n = Console.ReadLine();

        //Gọi biến và phương thức static
        Employee4.employeeCounter = Int32.Parse(n);
        Employee4.AddEmployee();

        // Hiển thị thông tin mới.
        Console.WriteLine($"Name: {e.name}");
        Console.WriteLine($"ID: {e.id}");
        Console.WriteLine($"New Number of Employees: {Employee4.employeeCounter}");
    }
}

```

Kết quả thực hiện:

```

/*
Input:
Matthias Berndt
AF643G
15
*
Sample Output:
Enter the employee's name: Matthias Berndt

```

```
Enter the employee's ID: AF643G
Enter the current number of employees: 15
Name: Matthias Berndt
ID:   AF643G
New Number of Employees: 16
*/
```

Tham khảo:

Beginning-C-Object-Oriented-Programming-2nd-Edition.pdf

<https://www.c-sharpcorner.com/UploadFile/ff2f08/understanding-polymorphism-in-C-Sharp/>

<https://www.c-sharpcorner.com/UploadFile/ff2f08/understanding-polymorphism-in-C-Sharp/>

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/properties>.

<http://www.srikanthtechnologies.com/books/csharp/oop1.html>

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/static-classes-and-static-class-members>

PHẦN 2. KẾ THỪA VÀ GIAO DIỆN

Trong phần trước, chúng ta đã học cách tạo các class, thêm thuộc tính và phương thức cũng như cách tạo các thể hiện (đối tượng) của các class. Bài học này sẽ giới thiệu các khái niệm về **kế thừa** (inheritance) và **tính đa hình** (polymorphism).

Kế thừa là một trong những tính năng cơ bản và mạnh mẽ nhất của bất kỳ ngôn ngữ OOP nào. Sử dụng kế thừa, bạn tạo các class cơ sở đóng gói chức năng chung. Các class khác có thể được dẫn xuất từ các class cơ sở này. Các class dẫn xuất kế thừa các thuộc tính và phương thức của các class cơ sở và mở rộng chức năng khi cần thiết.

Một tính năng OOP cơ bản thứ hai là tính đa hình. Tính đa hình cho phép một class cơ sở xác định các phương thức phải được thực hiện bởi bất kỳ class dẫn xuất nào. Lớp cơ sở xác định chữ ký thông báo mà các class dẫn xuất phải tuân theo, nhưng mã thực thi của phương thức **được để lại** cho class dẫn xuất. Sức mạnh của tính đa hình nằm ở chỗ các chúng ta có thể triển khai các phương thức của các class thuộc kiểu cơ sở theo cùng một kiểu. Mặc dù quy trình xử lý nội bộ của phương thức có thể khác nhau, nhưng người dùng sẽ biết dữ liệu vào (input) và dữ liệu ra (output) của các phương thức là giống nhau.

Trong bài học này, chúng ta sẽ tìm hiểu các vấn đề sau:

- Cách tạo và sử dụng các class cơ sở.
- Cách tạo và sử dụng các class dẫn xuất.
- Cách sử dụng phạm vi truy cập để kiểm soát tính kế thừa.
- Cách ghi đè các phương thức của class cơ sở.
- Cách triển khai các giao diện.
- Cách triển khai tính đa hình thông qua kế thừa và thông qua các giao diện.

1. Kế thừa (Inheritance)

1.1. Tạo class cơ sở và class dẫn xuất

Mục đích của kế thừa là tạo ra một class cơ sở đóng gói các thuộc tính và phương thức có thể được sử dụng bởi các class dẫn xuất cùng kiểu. Ví dụ, chúng ta có class Person từ bài học trước, trong class này có các thuộc tính id, name, age, và đây là class cơ sở. Bây giờ bạn có thể tạo 2 class dẫn xuất từ class cơ sở là class Nhân viên (Employee) và class Sinh viên (Student). Class Employee và Student sẽ có thêm các thuộc tính và phương thức riêng để mô tả một nhân viên hay một sinh viên.

Đoạn mã như sau:

```
class Employee : Person
{
    public double salary { get; set; }
    public double commistion { get; set; }
    public string department { get; set; }
```

```
public double total
{
    get
    {
        return salary + commistion;
    }
}
```

Class Employee có thuộc tính total, chỉ cho phép lấy ra giá trị (get), không cho thiết lập (set) giá trị khác vì tổng tiền (total) được tính bằng lương (salary) cộng với thưởng (commission).

```
class Student : Person
{
    public string email { get; set; }
    public int mark { get; set; }
    public char grade
    {
        get
        {
            if (mark >= 8)
                return 'A';
            else if (mark >= 6)
                return 'B';
            else if (mark >= 4)
                return 'C';
            else
                return 'D';
        }
    }
}
```

Class Student có thuộc tính grade (xếp loại), chỉ cho phép lấy ra giá trị (get), không cho thiết lập (set) giá trị khác vì grade được tính ra dựa trên điểm (mark) của sinh viên.

1.2. Sử dụng quyền truy cập trong class

Phạm vi truy cập	Mô tả
public	Các thành viên được khai báo là public có thể được truy cập ở bất cứ vị trí nào trong cùng một class hay class khác.
private	Các thành viên chỉ được truy cập trong cùng một class.
protected	Các thành viên chỉ có thể được truy cập trong cùng một class hoặc trong một class khác được dẫn xuất từ class đó.
internal	Cho phép các thành viên class chỉ có thể truy cập được trong các class của cùng một assembly (hợp ngữ). Hợp ngữ là một tệp được trình biên dịch tự động tạo khi biên dịch thành công ứng dụng .NET.

Khi thiết lập phân cấp class bằng cách sử dụng kế thừa, bạn phải quản lý cách các thuộc tính và phương thức của các class của bạn được truy cập. Hai quyền truy cập mà chúng ta hay sử dụng là công khai (public) và riêng tư (private). Nếu một phương thức hoặc thuộc tính của class cơ sở được hiển thị dưới dạng public, thì nó có thể được truy cập

bởi cả class dẫn xuất và bất kỳ ứng dụng nào của class dẫn xuất. Nếu bạn đặt thuộc tính hoặc phương thức của class cơ sở là private, thì class dẫn xuất hoặc các class khác sẽ không thể truy cập trực tiếp vào thuộc tính hoặc phương thức đó.

Bạn có thể muốn hiển thị một thuộc tính hoặc phương thức của class cơ sở cho một class dẫn xuất, nhưng không muốn hiển thị cho class khác ngoài class dẫn xuất. Trong trường hợp này, chúng ta sử dụng từ khóa protected (truy cập được bảo vệ).

Ví dụ quyền truy cập public với thuộc tính:

```
class User
{
    public string Name;
    public string Location;
    public int Age;
    public void GetUserDetails()
    {
        Console.WriteLine("Name: {0}", Name);
        Console.WriteLine("Location: {0}", Location);
        Console.WriteLine("Age: {0}", Age);
    }
}
class Program
{
    static void Main(string[] args)
    {
        User u = new User();
        u.Name = "Suresh Dasari";
        u.Location = "Hyderabad";
        u.Age = 32;
        u.GetUserDetails();
        Console.WriteLine("\nPress Enter Key to Exit..");
        Console.ReadLine();
    }
}
```

Chúng ta thấy 3 thuộc tính Name, Location, Age được khai báo là public, vì thế trong class Program, chúng ta có thể truy cập trực tiếp vào 3 biến này và gán các giá trị. Đặt thuộc tính của class là public là một điều không nên vì vi phạm tính đóng gói của lập trình hướng đối tượng – muốn truy cập vào thuộc tính phải thông qua set, get.

Ví dụ quyền truy cập protected với thuộc tính:

```
class Star
{
    protected int row;
    protected int column;
    public void Display()
    {
        for (int i = 0; i < row; i++)
        {
            for (int j = 0; j < column; j++)
            {
                Console.Write("* ");
            }
        }
    }
}
```

```

        Console.WriteLine();
    }
}
}
class PrintStar : Star
{
    static void Main(string[] args)
    {
        PrintStar st = new PrintStar();
        st.row = 7;
        st.column = 8;
        st.Display();
    }
}

```

Class PrintStar kế thừa từ class Star, hai thuộc tính row và column được khai báo là protected nên từ class PrintStar, chúng ta có thể sử dụng trực tiếp hai biến này.

1.3. Ghi đè các phương thức của một class cơ sở

Khi một class dẫn xuất kế thừa một phương thức từ một class cơ sở, nó sẽ kế thừa việc triển khai phương thức đó. Là người thiết kế class cơ sở, bạn có thể muốn để **một class dẫn xuất triển khai phương thức theo cách riêng của nó**. Điều này được gọi là ghi đè (overriding) phương thức class cơ sở.

Theo mặc định, một class dẫn xuất không thể ghi đè mã triển khai của class cơ sở của nó. Để cho phép một phương thức class cơ sở được ghi đè, bạn phải đưa từ khóa virtual vào định nghĩa phương thức. Trong class dẫn xuất, bạn xác định một phương thức có cùng chữ ký phương thức và cho biết nó đang ghi đè một phương thức class cơ sở bằng từ khóa override.

Ví dụ 1:

```

class TestOverride
{
    public class Employee
    {
        public string name;

        // Thuộc tính basepay is được định nghĩa là được bảo vệ,
        // vì vậy nó chỉ có thể được truy cập bởi class này và các class dẫn xuất.

        protected decimal basepay;

        // Phương thức khởi tạo thiết lập giá trị name và basepay cho employee.
        public Employee(string name, decimal basepay)
        {
            this.name = name;
            this.basepay = basepay;
        }

        // Khai báo là virtual, do đó phương thức này cần được ghi đè.
        public virtual decimal CalculatePay()
        {
            return basepay;
        }
    }

    // Tạo class mới kế thừa từ class Employee.
}

```



```

public class SalesEmployee : Employee
{
    // Tạo thêm thuộc tính mới cho class này.
    private decimal salesbonus;

    // Phương thức khởi tạo gọi phương thức khởi tạo của base-class
    // và gán giá trị cho biến salesbonus.
    public SalesEmployee(string name, decimal basepay,
        decimal salesbonus) : base(name, basepay)
    {
        this.salesbonus = salesbonus;
    }

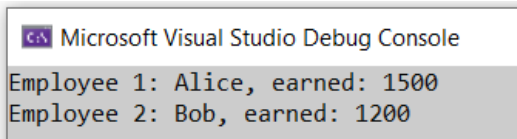
    // Ghi đè phương thức CalculatePay
    public override decimal CalculatePay()
    {
        return basepay + salesbonus;
    }
}

static void Main()
{
    // Tạo 2 đối tượng từ 2 class khác nhau.
    var employee1 = new SalesEmployee("Alice", 1000, 500);
    var employee2 = new Employee("Bob", 1200);

    // Mỗi class sẽ gọi phương thức CalculatePay của riêng mình.
    Console.WriteLine("Employee 1: " + employee1.name + ", earned: " +
employee1.CalculatePay());
    Console.WriteLine("Employee 2: " + employee2.name + ", earned: " +
employee2.CalculatePay());
}
}

```

Kết quả thực hiện:



```

Microsoft Visual Studio Debug Console
Employee 1: Alice, earned: 1500
Employee 2: Bob, earned: 1200

```

Ví dụ 2: Class Rectangle và Class Box.

```

class Rectangle
{
    public int height { get; set; }
    public int width { get; set; }

    public Rectangle()
    {
    }
    public Rectangle(int h, int w)
    {
        height = h;
        width = w;
    }

    public virtual void Input()
    {
        Console.WriteLine("=== Input information ===");
        Console.Write("Heigh: ");
    }
}

```

```

        height = int.Parse(Console.ReadLine());
        Console.Write("Width: ");
        width = int.Parse(Console.ReadLine());
    }
    public virtual void Display()
    {
        Console.WriteLine("=== Output information ===");
        Console.WriteLine("Heigh: " + height);
        Console.WriteLine("Width: " + width);
    }
    public int Perimeter()
    {
        return (height + width) * 2;
    }
    public int Area()
    {
        return height * width;
    }
    public void SayGoodbye() {
        Console.WriteLine("Goodbye from Rectangle !");
    }
}

```

Class Box:

```

class Box : Rectangle
{
    public int depth { get; set; }
    public Box()
    {
    }
    public Box(int h, int w, int d) : base(h, w)
    {
        depth = d;
    }
    public override void Input()
    {
        base.Input();
        Console.Write("Depth: ");
        depth = int.Parse(Console.ReadLine());
    }
    public override void Display()
    {
        base.Display();
        Console.WriteLine("Depth: " + depth);
    }
    public int Volume()
    {
        return base.Area()* depth;
    }
    public void SayHello()
    {
        Console.WriteLine("Hello world");
    }
}

```

Test sự hoạt động của 2 class trên:

```
static void Main(string[] args)
{
    //TEST BOX, RECTANGLE
    Rectangle r = new Rectangle(4,5);
    r.Display();
    Console.WriteLine("Area is: "+r.Area());

    Box b = new Box(4, 5, 3);
    b.Display();
    Console.WriteLine("Area is: " + b.Area());
    b.SayHello();

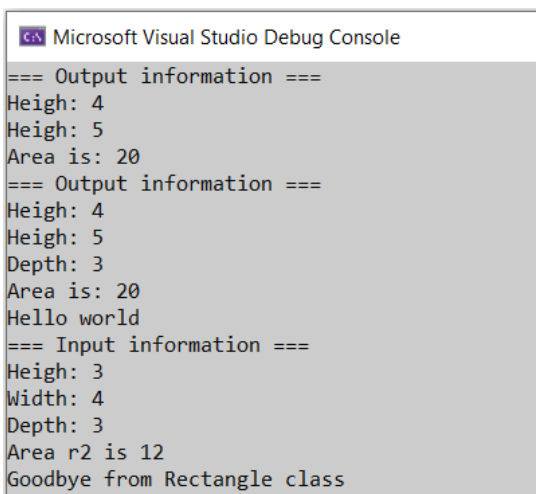
    Rectangle r2 = new Box();
    r2.Input();
    int c = r2.Volume(); //error, vì trong Rectangle không có Cube
    //(trong Box có)
    Console.WriteLine("Area r2 is "+r2.Area());

    r2.SayGoodbye(); //Ok, trong Box không có sayGoodbye nhưng
    //trong Rectangle có, nên sẽ gọi từ class base.

    r2.sayHello(); //error vì trong Rectangle không có sayHello (trong Box có)

    Box x2 = new Rectangle(); //error
```

Kết quả thực hiện:



```
Microsoft Visual Studio Debug Console
=== Output information ===
Heigh: 4
Heigh: 5
Area is: 20
=== Output information ===
Heigh: 4
Heigh: 5
Depth: 3
Area is: 20
Hello world
=== Input information ===
Heigh: 3
Width: 4
Depth: 3
Area r2 is 12
Goodbye from Rectangle class
```

*Ghi nhớ:

Khi một phương thức ảo (virtual) được gọi, trong thời gian chạy, sẽ kiểm tra để tìm thành viên ghi đè (override). Thành viên ghi đè trong lớp dẫn xuất được gọi nếu có, còn trường hợp không có lớp dẫn xuất nào ghi đè thành viên thì thành viên ban đầu sẽ được gọi.

Theo mặc định, các phương thức là không ảo (không là virtual). Bạn không thể ghi đè một phương thức không phải ảo.

Không thể sử dụng từ khóa virtual với các từ khóa static, abstrac, private, override.

Sau đây là ví dụ sử dụng một thuộc tính ảo:

```
class MyBaseClass
{
```

```
// virtual auto-implemented property. Overrides can only
// provide specialized behavior if they implement get and set accessors.
public virtual string Name { get; set; }

// ordinary virtual property with backing field
private int num;
public virtual int Number
{
    get { return num; }
    set { num = value; }
}
}

class MyDerivedClass : MyBaseClass
{
    private string name;

    // Override auto-implemented property with ordinary property
    // to provide specialized accessor behavior.
    public override string Name
    {
        get
        {
            return name;
        }
        set
        {
            if (!string.IsNullOrEmpty(value))
            {
                name = value;
            }
            else
            {
                name = "Unknown";
            }
        }
    }
}
}
```

1.3.1. Gọi phương thức từ class dẫn xuất từ base class

Trong ví dụ trên, chúng ta có thể sử dụng:

```
Rectangle r2 = new Box();
```

Từ đối tượng r2, chúng ta sẽ gọi các phương thức. Vậy phương thức nào sẽ được thực thi trong trường hợp cả 2 class này đều chứa các phương thức có tên giống nhau.

Trong ví dụ này, bạn sẽ thấy các phương thức trong Box sẽ được thực thi với điều kiện phương thức đó có trong class Rectangle, chúng ta vẫn nhập được dữ liệu đủ cho 3 thuộc tính, nhưng khi gọi Area() thì kết quả nhận được là tính diện tích, vì trong class Box không có phương thức Area() nên nó sẽ tự động gọi Area() từ class base. Tương tự method SayGoodbye có trong Rectangle nhưng không có trong Box nên sẽ phương thức này được gọi từ class dẫn xuất.

1.3.2. Gọi phương thức của base class từ phương thức dẫn xuất.

*Gọi phương thức khởi tạo từ base class

Sử dụng: `:base(danh sách tham số).`

```
public Box(int h, int w, int d) : base(h, w)
{
    depth = d;
}
```

*Gọi phương thức từ base class

Sử dụng: `base.tên phương thức().`

```
public override void Input()
{
    base.Input();
    Console.WriteLine("Depth: ");
    depth = int.Parse(Console.ReadLine());
}
```

1.3.3. Sử dụng từ khoá new với phương thức

Trong lớp dẫn xuất, có phương thức tên trùng với tên của phương thức cơ sở, mà phương thức này trong lớp cơ sở không có từ khoá *abstract* hay *virtual* thì phải sử dụng từ khoá *new* để che giấu phương thức.

Ví dụ:

```
class Person
{
    public void Input()
}
class Student:Person
{
    public new void Input()
}
```

1.3.4. Ghi đè phương thức ToString()

Là phương thức trả về một chuỗi đại diện cho các đối tượng hiện tại.

```
public virtual string? ToString ();
```

Chúng ta xem xét class Person sau đây:

```
class Person
{
    public string id { get; set; }
    public string name { get; set; }
    public int age { get; set; }
    public Person()
    {
        id = "DEFAULT_ID";
        name = "DEFAULT_NAME";
    }
}
```

```

        age = 0;
    }
    public Person(string id)
    {
        this.id = id;
    }
    public Person(string id, string name, int age)
    {
        this.id = id;
        this.name = name;
        this.age = age;
    }
}

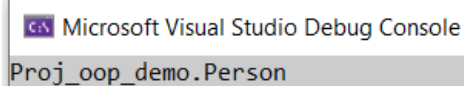
```

Trong phương thức main:

```

Person p = new Person("p01", "Nguyen Thi Lan", 20);
Console.WriteLine(p);

```



Microsoft Visual Studio Debug Console
Proj_oop_demo.Person

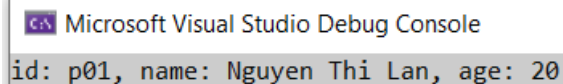
Bổ sung thêm phương thức ToString vào class Person như sau:

```

public override string ToString()
{
    return "id: " + id + ", name: " + name + ", age: " + age;
}

```

Kết quả hiển thị:



Microsoft Visual Studio Debug Console
id: p01, name: Nguyen Thi Lan, age: 20

Như vậy, thông tin chi tiết về Person p được hiển thị thay vì tên Class được hiển thị.

1.3.5. Ghi đề phương thức Equal()

Là phương thức xác định xem hai đối tượng có bằng nhau hay không.

```

public virtual bool Equals (object? obj);

```

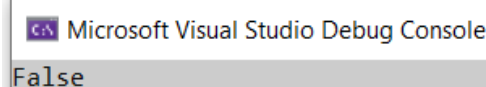
Tiếp theo ví dụ trên, chúng ta so sánh 2 đối tượng:

```

Person p1 = new Person("p01", "Nguyen Thi Lan", 20);
Person p2 = new Person("p01", "Nguyen Thi Lan", 20);
Console.WriteLine(p1.Equals(p2));

```

Đối tượng p1 và p2 có các thông tin trùng nhau hoàn toàn. Vậy kết quả của sự so sánh này là gì?



Microsoft Visual Studio Debug Console
False

Trong lập trình hướng đối tượng, hai đối tượng không bao giờ bằng nhau trừ khi hai đối tượng đó là một.

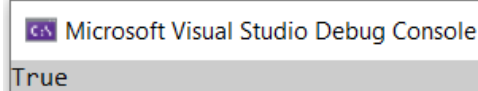
Vì mỗi khi tạo một đối tượng mới, sẽ có một vùng nhớ để lưu trữ đối tượng đó. Và khi so sánh các đối tượng với nhau, thực chất là sự so sánh hai vị trí trong bộ nhớ (reference) và như vậy các đối tượng không bao giờ là bằng nhau.

Bằng cách viết lại phương thức *Equals*, chúng ta có thể thay đổi điều này; tùy theo tiêu chí muốn so sánh.

Với ví dụ trên, để so sánh hai đối tượng thì chúng ta sẽ so sánh theo id. Nếu id là bằng nhau thì hai đối tượng đó là bằng nhau.

Trong class *Person*, viết thêm phương thức *Equals* như sau:

```
public override bool Equals(object obj)
{
    Person p = (Person)obj;
    return (this.id.Equals(p.id));
}
```



*Lưu ý:

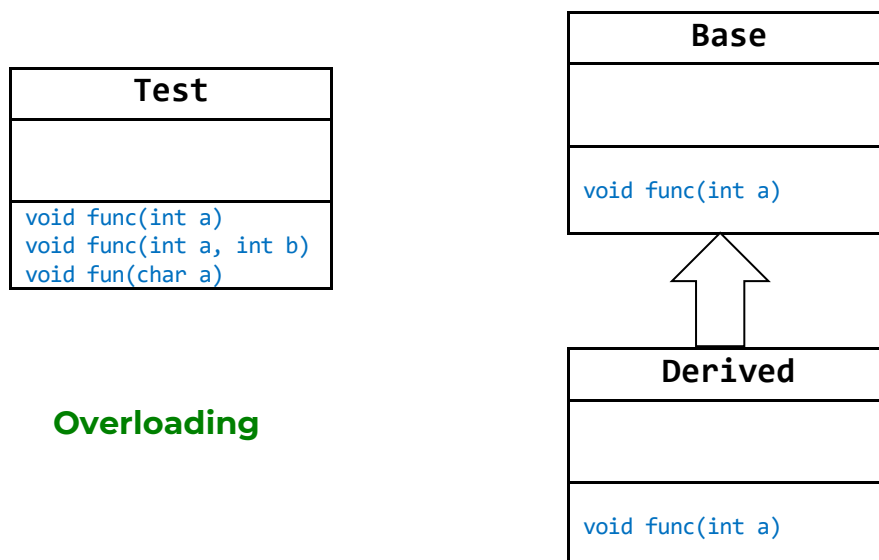
Trong class base, chỉ phương thức có khai báo *virtual* là được ghi đè. Do vậy, nếu không muốn phương thức bị ghi đè trong các class dẫn xuất, chỉ cần bỏ từ khóa *virtual*.

1.4. Tính đa hình trong lập trình hướng đối tượng.

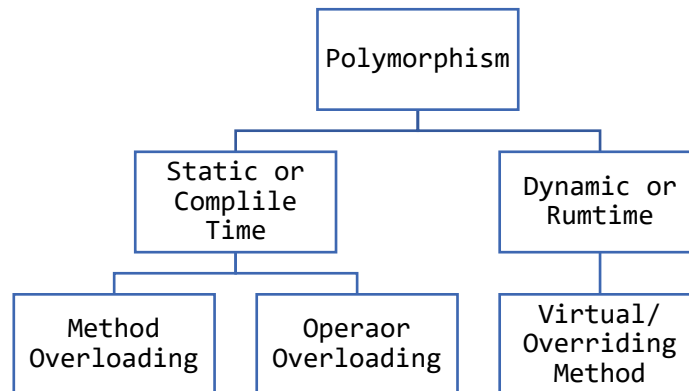
Chúng ta có 2 khái niệm: overriding và overloading

Overloading (nạp chồng) là về cùng một phương thức nhưng có các chữ ký (signatures) khác nhau.

Overriding (ghi đè) là dùng cùng một phương thức, cùng một chữ ký (signatures) nhưng ở các lớp khác nhau được kết nối thông qua kế thừa.



Overriding



Static or Compile Time: Nạp chồng (overloading) phương thức là một ví dụ về Đa hình tĩnh. Trong quá trình nạp chồng, phương thức / hàm có cùng tên nhưng khác chữ ký. Nó còn được gọi là Đa hình thời gian biên dịch vì quyết định gọi phương thức nào được thực hiện tại thời điểm biên dịch. Overloading là khái niệm trong đó các tên phương thức giống nhau với một tập các tham số khác nhau về thứ tự, số lượng hay kiểu.

Tính đa hình thời gian tĩnh hoặc thời gian biên dịch được minh họa thông qua một quá trình gọi là quá tải (**overloading**). Bạn có thể triển khai các phương thức khác nhau của một đối tượng có cùng tên. Sau đó, đối tượng có thể cho biết phương thức nào sẽ triển khai tùy thuộc vào ngữ cảnh (nói cách khác, số lượng và kiểu đối số được truyền vào) của thông điệp. Ví dụ: bạn có thể tạo hai phương thức của một đối tượng hàng tồn kho để tra cứu giá của một sản phẩm. Cả hai phương thức này sẽ được đặt tên là getPrice. Một đối tượng khác có thể gọi phương thức này và chuyển tên của sản phẩm hoặc ID sản phẩm. Đối tượng khoảng không quảng cáo có thể cho biết phương thức getPrice nào sẽ chạy bằng cách chuyển giá trị chuỗi hay giá trị số nguyên cùng với yêu cầu. Trong ví dụ trên phương thức Add() nào được gọi sẽ phụ thuộc vào tham số truyền vào là 2 hay 3 số.

*Ví dụ về overloading:

Class TestData có hai phương thức có cùng tên "Add" nhưng với các tham số đầu vào khác nhau (phương thức đầu tiên có ba tham số và phương thức thứ hai có hai tham số).

```

public class TestData
{
    public int Add(int a, int b, int c)
    {
        return a + b + c;
    }
    public int Add(int a, int b)
    {
        return a + b;
    }
}
  
```

```
class Program
{
    static void Main(string[] args)
    {
        TestData dataClass = new TestData();
        int add2 = dataClass.Add(45, 34, 67);
        int add1 = dataClass.Add(23, 34);
    }
}
```

Dynamic hoặc Runtime: Đa hình động / thời gian chạy còn được gọi là liên kết trễ. Ở đây, tên phương thức và chữ ký phương thức (số lượng tham số và kiểu tham số phải giống nhau và có thể có cách triển khai khác). Ghi đè (overriding) phương thức là một ví dụ về tính đa hình động.

Ghi đè phương thức có thể được thực hiện bằng cách sử dụng kế thừa. Với việc ghi đè phương thức, lớp cơ sở và lớp dẫn xuất có thể có cùng tên phương thức và giống nhau. Trình biên dịch sẽ không nhận thức được phương thức có sẵn để ghi đè chức năng, vì vậy trình biên dịch không gây ra lỗi tại thời điểm biên dịch. Trình biên dịch sẽ quyết định phương thức nào sẽ gọi trong thời gian chạy và nếu không tìm thấy phương thức đó thì sẽ thông báo lỗi.

*Ví dụ về Overriding

```
public class Drawing
{
    public virtual double Area()
    {
        return 0;
    }
}
public class Circle : Drawing
{
    public double Radius { get; set; }
    public Circle()
    {
        Radius = 5;
    }
    public override double Area()
    {
        return (3.14) * Math.Pow(Radius, 2);
    }
}
public class Square : Drawing
{
    public double Length { get; set; }
    public Square()
    {
        Length = 6;
    }
    public override double Area()
    {
        return Math.Pow(Length, 2);
    }
}
```

```
public class Rectangle : Drawing
{
    public double Height { get; set; }
    public double Width { get; set; }
    public Rectangle()
    {
        Height = 5.3;
        Width = 3.4;
    }
    public override double Area()
    {
        return Height * Width;
    }
}
class Program
{
    static void Main(string[] args)
    {
        Drawing circle = new Circle();
        Console.WriteLine("Area : " + circle.Area());

        Drawing square = new Square();
        Console.WriteLine("Area : " + square.Area());

        Drawing rectangle = new Rectangle();
        Console.WriteLine("Area : " + rectangle.Area());
    }
}
```

1.5. Lớp niêm phong (sealed class)

Theo mặc định, bất kỳ class nào trong C# cũng có thể được kế thừa. Khi tạo các class có thể được kế thừa, bạn phải chú ý rằng chúng không được sửa đổi theo cách mà các class dẫn xuất không còn hoạt động như dự kiến. Nếu không cẩn thận, bạn có thể tạo ra các chuỗi kế thừa phức tạp khó quản lý và gỡ lỗi. Ví dụ: giả sử bạn tạo một class Employee dẫn xuất dựa trên class Person. Một lập trình viên khác có thể đến và tạo một class dẫn xuất dựa trên Employee và sử dụng nó theo những cách mà bạn không mong muốn. Bằng cách sử dụng từ khóa sealed khi tạo class, bạn sẽ ngăn chặn được việc class này có thể là nguồn gốc cho các class khác, tức là ngăn cho một class không thể được kế thừa.

Câu lệnh sau làm cho class Employee được niêm phong, không thể kế thừa:

```
sealed class Employee : Person
```

*Ví dụ

```
using System;
class Class1
{
    static void Main(string[] args)
    {
        TestNumber ob = new TestNumber();
        int total = ob.Add(4, 5);
        Console.WriteLine("Total = " + total.ToString());
    }
}
```

```
sealed class TestNumber // Lớp niêm phong
{
    public int Add(int x, int y)
    {
        return x + y;
    }
}
```

1.6. Lớp trừu tượng (class abstract)

Một class trừu tượng là một class không hoàn chỉnh hoặc một class đặc biệt mà chúng ta **không thể khởi tạo được**. Mục đích của một class trừu tượng là cung cấp một bản thiết kế cho các class dẫn xuất và thiết lập một số quy tắc mà các class dẫn xuất phải thực hiện khi chúng kế thừa một class trừu tượng.

Chúng ta có thể sử dụng một class trừu tượng làm class cơ sở và tất cả các class dẫn xuất phải triển khai các định nghĩa trừu tượng. Một phương thức trừu tượng phải được triển khai trong tất cả các class không trừu tượng bằng cách sử dụng từ khóa override. Sau khi ghi đè phương thức trừu tượng nằm trong class không trừu tượng. Chúng ta có thể dẫn xuất class này trong một class khác và một lần nữa chúng ta có thể ghi đè cùng một phương thức trừu tượng với nó.

Để khai báo lớp trừu tượng, chúng ta sử dụng từ khóa **abstract** trước từ khoá **class** như sau:

```
abstract class <tên_class>
{
    // Các mã bên trong
}
```

Lưu ý: Một phương thức trừu tượng chỉ được khai báo trong lớp trừu tượng.

Có thể khai báo một phương thức là virtual trong lớp trừu tượng.

Ví dụ:

```
abstract class Example
{
    protected virtual string SayHello(string)
    {
        //Các mã bên trong
    }
}
```

*Các đặc điểm của class trừu tượng của C

- Một class trừu tượng có thể kế thừa từ một class và một hoặc nhiều giao diện.
- Một class trừu tượng có thể có các phương thức không trừu tượng.
- Một class trừu tượng có thể có các phương thức, thuộc tính,... là trừu tượng
- Một class trừu tượng có thể có hằng số và trường.
- Một class trừu tượng có thể triển khai một thuộc tính.
- Một class trừu tượng có thể có phương thức khởi tạo hoặc hủy.
- Một class trừu tượng không thể được kế thừa bởi các cấu trúc.
- Một class trừu tượng không thể hỗ trợ đa kế thừa.

Ví dụ:

```
abstract class Shape
{
    public abstract int GetArea();
    public void Output()
    {
        Console.WriteLine("Hello World");
    }
}
class Square : Shape
{
    public int side{ get; set; }

    public Square(int side)
    {
        this.side = side;
    }

    // GetArea method is required to avoid a compile-time error.
    public override int GetArea()
    {
        return side * side;
    }

    static void Main()
    {
        var sq = new Square(12);
        Console.WriteLine($"Area of the square = {sq.GetArea()}");
    }
}
```

// Hiển thị: Area of the square = 144

Với class Shape chúng ta không thể khởi tạo:

```
Shape sh = new Shape();
```

Class Square kế thừa từ class Shape, bắt buộc phải viết lại phương thức `GetArea()`.

2. Giao diện (interface)

Như đã thấy trước đó, chúng ta có thể tạo một class cơ sở trừu tượng mà không chứa bất kỳ đoạn mã cụ thể nào, tức là trong class này chỉ chứa các phương thức, thuộc tính ... là trừu tượng. Khi sử dụng một class trừu tượng, các class dẫn xuất từ nó phải thực thi các phương thức kế thừa của nó. Chúng ta có thể sử dụng một kỹ thuật khác để đạt được kết quả tương tự. Trong trường hợp này, thay vì xác định một class trừu tượng, chúng ta xác định một giao diện (interface) xác định các chữ ký của phương thức.

Các class thực thi interface được yêu cầu phải viết lại tất cả các định nghĩa chữ ký trong interface và không thể thay đổi định nghĩa đó. Kỹ thuật này hữu ích để đảm bảo rằng khi sử dụng các class, chúng ta sẽ biết phương thức nào có sẵn, cách gọi chúng và giá trị trả về mong đợi. Giống như trước khi thực hiện một nhiệm vụ nào đó, chúng ta sẽ lập ra một bản kế hoạch để đạt được mục tiêu, sau đó từng bước triển khai kế hoạch.

2.1. Khai báo

```
interface <Tên_interface>
{
    Các thuộc tính;
    Các phương thức;
}
```

Thực thi giao diện:

```
class <Tên_class> : <Tên_interface>
{
    //Chi tiết mã cho các thuộc tính, phương thức.
}
```

Chú ý : Một class có thể được thừa kế từ nhiều interface. Nếu các interface có các phương thức trùng nhau thì trong class thực thi, chúng ta gọi theo cách: tên interface và tên phương thức: TenInterface.TenPhuongthuc.

Ví dụ 1: Một lớp thực thi 2 giao diện.

```
interface IClass1
{
    string getName();
}
interface IClass2
{
    string getName();
    void SayHello();
}
class Example : IClass1, IClass2
{
    string IClass1.getName()
    {
        // Đoạn mã cho phương thức
    }
    string IClass2.getName()
    {
        // Đoạn mã cho phương thức
    }
    void SayHello()
    {
        // Đoạn mã cho phương thức
    }
}
```

Ví dụ 2: Tạo interface thao tác với file.

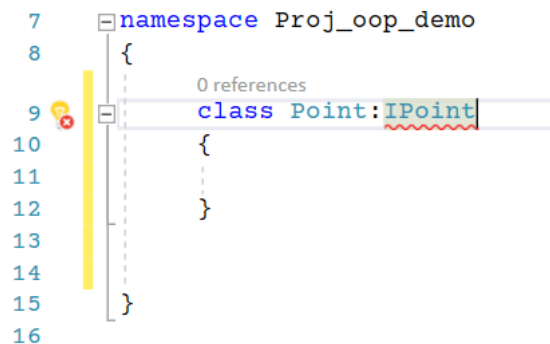
```
interface IFile
{
    void ReadFile();
    void WriteFile(string text);
}
```

Tên interface nên bắt đầu bằng ký tự I để chúng ta dễ nhận biết trong project.

2.2. Ví dụ tạo giao diện và thực thi

```
interface IPoint {
    public int x{ get; set; }
    public int y { get; set; }
    public double distance {get;}
    public void InputXy();
    public void OutputXy();
}
```

Khi tạo class thực thi IPoint, sẽ xuất hiện cảnh báo yêu cầu thực thi tất cả các phương thức trong interface IPoint:



```
7 namespace Proj_oop_demo
8 {
9     class Point:IPoint
10    {
11    }
12 }
13
14
15
16
```

Click vào hình ảnh bóng đèn vàng, và chọn mục Implement Interface, sẽ tự động xuất hiện các phương thức:

```
class Point : IPoint
{
    public int x { get => throw new NotImplementedException(); set => throw new
NotImplementedException(); }
    public int y { get => throw new NotImplementedException(); set => throw new
NotImplementedException(); }

    public double distance => throw new NotImplementedException();

    public void InputXy()
    {
        throw new NotImplementedException();
    }
    public void OutputXy()
    {
        throw new NotImplementedException();
    }
}
```

Chúng ta viết lại các phương thức trên:

```
class Point : IPoint
{
    public int x { get; set; }
    public int y { get; set;}

    public double distance
    {
        get { return (x * x + y * y); }
    }
}
```



```

    }

    public void InputXy()
    {
        Console.WriteLine("==== INPUT ===");
        Console.Write("Input x: ");
        x = int.Parse(Console.ReadLine());
        Console.Write("Input y: ");
        y = int.Parse(Console.ReadLine());
    }
    public void Output()
    {
        Console.WriteLine("=== OUTPUT===");
        Console.WriteLine("Distance between "+x+" and "+y+" is: "+distance);
    }
}

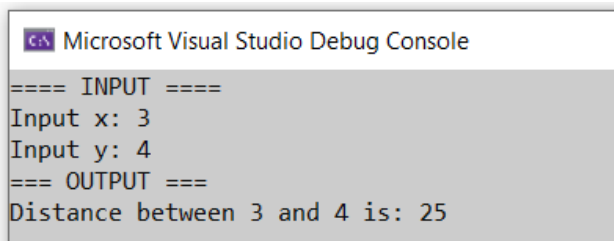
```

Trong phương thức Main()

```

Point p = new Point();
p.InputXy();
p.Output();

```



```

Microsoft Visual Studio Debug Console
==== INPUT ====
Input x: 3
Input y: 4
=== OUTPUT ===
Distance between 3 and 4 is: 25

```

2.3. Giao diện IComparable

Xác định phương pháp so sánh tổng quát mà một kiểu giá trị hoặc lớp thực hiện để tạo ra một phương pháp so sánh dành riêng cho kiểu để sắp xếp hoặc sắp xếp các thể hiện của nó.

IComparable được thực hiện bởi các loại có giá trị có thể được sắp xếp hoặc sắp xếp. Interface này sẽ yêu cầu class thực thi nó phải viết đề phương thức CompareTo. Phương thức CompareTo sẽ được tự động thực thi bởi các phương thức như Array.Sort và ArrayList.Sort.

Chúng ta xem xét các ví dụ dưới đây:

Ví dụ 1: Sử dụng List<string> và gọi phương thức Sort() để sắp xếp dữ liệu:

Để sử dụng List, chúng ta cần khai báo `using System.Collections.Generic`

```

static void Main(string[] args)
{
    List<string> li = new List<string>();
    li.Add("Lan");
    li.Add("Tuan");
    li.Add("Huong");
    li.Add("Van");
    li.Add("Binh");
    Console.WriteLine("=== DANH SACH TRUOC KHI SAP XEP ===");
}

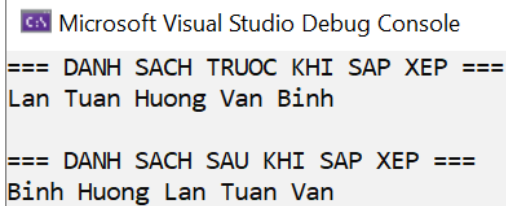
```

```

foreach (var item in li)
{
    Console.Write(item + " ");
}
Console.WriteLine("\n");
li.Sort();
Console.WriteLine("=== DANH SACH SAU KHI SAP XEP ===");
foreach (var item in li)
{
    Console.Write(item + " ");
}
Console.WriteLine("\n");
}

```

Kết quả hiển thị:



```

Microsoft Visual Studio Debug Console
=== DANH SACH TRUOC KHI SAP XEP ===
Lan Tuan Huong Van Binh

=== DANH SACH SAU KHI SAP XEP ===
Binh Huong Lan Tuan Van

```

Ví dụ 2: Sử dụng List<int> và gọi phương thức sort() để sắp xếp dữ liệu:

```

static void Main(string[] args)
{
    List<int> li = new List<int>();
    li.Add(100);
    li.Add(5);
    li.Add(80);
    li.Add(30);
    li.Add(60);
    Console.WriteLine("=== DANH SACH TRUOC KHI SAP XEP ===");
    foreach (var item in li)
    {
        Console.Write(item + " ");
    }
    Console.WriteLine("\n");
    li.Sort();
    Console.WriteLine("=== DANH SACH SAU KHI SAP XEP ===");
    foreach (var item in li)
    {
        Console.Write(item + " ");
    }
    li.Reverse();
    Console.WriteLine("\n");
    Console.WriteLine("=== DANH SACH SAU KHI DAO NGUOC ===");
    foreach (var item in li)
    {
        Console.Write(item + " ");
    }
    Console.WriteLine("\n");
}

```

```

Microsoft Visual Studio Debug Console

=== DANH SACH TRUOC KHI SAP XEP ===
Lan Tuan Huong Van Binh

=== DANH SACH SAU KHI SAP XEP ===
Binh Huong Lan Tuan Van

=== DANH SACH SAU KHI DAO NGUOC ===
Van Tuan Lan Huong Binh

```

Như vậy với kiểu dữ liệu trong `List<type>`, với `type` là kiểu dữ liệu nguyên thủy, thì phương thức `Sort()` hoạt động tốt.

Ví dụ 3: Sử dụng `List` với kiểu dữ liệu là đối tượng.

Chúng ta có class `Employee` có cấu trúc:

```

class Employee
{
    public string id{ get; set; }
    public string name{ get; set; }
    public int salary { get; set; }

    public Employee(string id, string name, int salary)
    {
        this.id = id;
        this.name = name;
        this.salary = salary;
    }
    public override string ToString()
    {
        return id + " " + name + " " + salary;
    }
}

```

Yêu cầu: Tạo danh sách có 5 employee, sắp xếp theo chiều tăng dần của lương.

Đoạn mã như sau:

```

static void Main(string[] args)
{
    List<Employee> li = new List<Employee>();
    li.Add(new Employee("e01", "Mai Anh", 5000));
    li.Add(new Employee("e02", "Van Tuan", 3000));
    li.Add(new Employee("e03", "Thu Hang", 4000));
    li.Add(new Employee("e04", "Cong Tien", 2500));
    li.Add(new Employee("e05", "Long Van", 4500));

    Console.WriteLine("=== DANH SACH BAN DAU LA ===");
    li.ForEach(emp => Console.WriteLine(emp));

    li.Sort();
    Console.WriteLine("\n");
    Console.WriteLine("=== DANH SACH SAU KHI SAP XEP LA ===");
    li.ForEach(emp => Console.WriteLine(emp));
}

```

Khi chạy chương trình, xuất hiện lỗi không thể so sánh được hai phần tử trong mảng:

```
Microsoft Visual Studio Debug Console

=== DANH SACH BAN DAU LA ===
e01 Mai Anh 5000
e02 Van Tuan 3000
e03 Thu Hang 4000
e04 Cong Tien 2500
e05 Long Van 4500
Unhandled exception. System.InvalidOperationException: Failed to compare two elements in the array.
--> System.ArgumentException: At least one object must implement IComparable.
   at System.Collections.Comparer.Compare(Object a, Object b)
   at System.Collections.Generic.ObjectComparer`1.Compare(T x, T y)
   at System.Collections.Generic.ArraySortHelper`1.InsertionSort(Span`1 keys, Comparison`1 comparer)
   at System.Collections.Generic.ArraySortHelper`1.Introsort(Span`1 keys, Int32 depthLimit, Comparison`1
   at System.Collections.Generic.ArraySortHelper`1.IntrospectiveSort(Span`1 keys, Comparison`1 comparer)
   at System.Collections.Generic.ArraySortHelper`1.Sort(Span`1 keys, IComparer`1 comparer)
   --- End of inner exception stack trace ---
   at System.Collections.Generic.ArraySortHelper`1.Sort(Span`1 keys, IComparer`1 comparer)
   at System.Array.Sort[T](T[] array, Int32 index, Int32 length, IComparer`1 comparer)
   at System.Collections.Generic.List`1.Sort(Int32 index, Int32 count, IComparer`1 comparer)
   at System.Collections.Generic.List`1.Sort()
```

Bởi vì chúng ta phải so sánh hai đối tượng để quyết định đối tượng nào đứng trước, đối tượng nào đứng sau. Nhưng một đối tượng có thể có nhiều thuộc tính, nên phải viết ra so sánh theo thuộc tính nào. Giống như chúng ta có một danh sách sinh viên, mà mỗi sinh viên có các thuộc tính như mã sinh viên, họ tên, điểm... và thường với danh sách này, nếu không có gì đặc biệt, sẽ được sắp xếp theo tên.

Để sửa lỗi cho ví dụ trên, chúng ta cần thực thi interface [IComparable](#), và viết lại phương thức CompareTo để quyết định so sánh theo tiêu chí nào.

Bổ sung vào class Employee:

```
class Employee:IComparable {
```

Viết thêm phương thức:

```
public int CompareTo(object obj)
{
    Employee emp = (Employee)obj;
    return (this.salary - emp.salary);
}
```

Chỉ cần trừ 2 giá trị salary là chúng ta biết emp nào lớn hơn hay nhỏ hơn.

Kết quả thực hiện, danh sách đã được sắp xếp tăng dần theo salary:

```
Microsoft Visual Studio Debug Console

=== DANH SACH BAN DAU LA ===
e01 Mai Anh 5000
e02 Van Tuan 3000
e03 Thu Hang 4000
e04 Cong Tien 2500
e05 Long Van 4500

=== DANH SACH SAU KHI SAP XEP LA ===
e04 Cong Tien 2500
e02 Van Tuan 3000
e03 Thu Hang 4000
e05 Long Van 4500
e01 Mai Anh 5000
```

Nếu muốn sắp xếp giảm dần theo salary, chỉ cần đảo lại phép trừ:

```
return (emp.salary - this.salary);
```

```
Microsoft Visual Studio Debug Console

=== DANH SACH BAN DAU LA ===
e01 Mai Anh 5000
e02 Van Tuan 3000
e03 Thu Hang 4000
e04 Cong Tien 2500
e05 Long Van 4500

=== DANH SACH SAU KHI SAP XEP LA ===
e01 Mai Anh 5000
e05 Long Van 4500
e03 Thu Hang 4000
e02 Van Tuan 3000
e04 Cong Tien 2500
```

Tại một thời điểm, chúng ta chỉ có thể sắp xếp theo một tiêu chí. Nếu muốn thay đổi, chúng ta sửa lại trong phương thức CompareTo(), ví dụ chúng ta muốn sắp xếp theo name của employee:

```
public int CompareTo(object obj)
{
    Employee emp = (Employee)obj;
    return (this.name.CompareTo(emp.name));
}
```

```
Microsoft Visual Studio Debug Console

=== DANH SACH BAN DAU LA ===
e01 Mai Anh 5000
e02 Van Tuan 3000
e03 Thu Hang 4000
e04 Cong Tien 2500
e05 Long Van 4500

=== DANH SACH SAU KHI SAP XEP LA ===
e04 Cong Tien 2500
e05 Long Van 4500
e01 Mai Anh 5000
e03 Thu Hang 4000
e02 Van Tuan 3000
```

2.4. Giao diện IComparer

Interface IComparer cũng được dùng để so sánh các đối tượng, nhưng sự sắp xếp này có thể tùy chọn, do đó cùng một danh sách đối tượng, chúng ta có thể sắp xếp được theo các tiêu chí khác nhau bằng cách tạo nhiều lớp thực thi giao diện IComparer.

IComparable so sánh đối tượng nội tại (inner) và một đối tượng khác, trong khi IComparer so sánh hai đối tượng.

Ví dụ 1, sử dụng ArrayList.

```
namespace Proj_oop_demo
{
    public class Car : IComparable
    {
        private int year;
        private string make;

        public Car(string Make, int Year)
        {
            make = Make;
            year = Year;
        }
        public int Year
        {
            get { return year; }
            set { year = value; }
        }
        public string Make
        {
            get { return make; }
            set { make = value; }
        }
    }
    // Bắt đầu viết class lồng nhau.
    // Class này được dùng để sắp xếp tăng dần các Car theo year
    private class sortYearAscendingHelper : IComparer
    {
        int IComparer.Compare(object a, object b)
        {
            Car c1 = (Car)a;
            Car c2 = (Car)b;

            if (c1.year > c2.year)
                return 1;

            if (c1.year < c2.year)
                return -1;

            else
                return 0;
        }
    }

    // Class này được dùng để sắp xếp giảm dần các Car theo year
    private class sortYearDescendingHelper : IComparer
    {
        int IComparer.Compare(object a, object b)
        {
            Car c1 = (Car)a;
            Car c2 = (Car)b;

            if (c1.year < c2.year)
                return 1;

            if (c1.year > c2.year)
                return -1;

            else
```

```

        return 0;
    }
}

// Class này được dùng để sắp xếp giảm dần các Car theo make.
private class sortMakeDescendingHelper : IComparer
{
    int IComparer.Compare(object a, object b)
    {
        Car c1 = (Car)a;
        Car c2 = (Car)b;
        return String.Compare(c2.make, c1.make);
    }
}

// Kết thúc các class bên trong.

// Thực thi IComparable để sắp xếp mặc định cho các đối tượng Car
int IComparable.CompareTo(object obj)
{
    Car c = (Car)obj;
    return String.Compare(this.make, c.make);
}

// Phương thức trả về đối tượng IComparer để sắp xếp.
public static IComparer sortYearAscending()
{
    return (IComparer)new sortYearAscendingHelper();
}

// Phương thức trả về đối tượng IComparer để sắp xếp.
public static IComparer sortYearDescending()
{
    return (IComparer)new sortYearDescendingHelper();
}

// Phương thức trả về đối tượng IComparer để sắp xếp.
public static IComparer sortMakeDescending()
{
    return (IComparer)new sortMakeDescendingHelper();
}
}
}

```

Đoạn mã trong phương thức Main như sau:

```

static void Main(string[] args)
{
    // Tạo một mảng đối tượng Car.
    Car[] arrayOfCars = new Car[6]
    {
        new Car("Ford",1992),
        new Car("Fiat",1988),
        new Car("Buick",1932),
        new Car("Ford",1932),
        new Car("Dodge",1999),
        new Car("Honda",1977)
    };

    // Hiển thị tiêu đề cho mảng.
}

```

```

Console.WriteLine("Array - Unsorted\n");

foreach (Car c in arrayOfCars)
    Console.WriteLine(c.Make + "\t\t" + c.Year);

// Sử dụng IComparable để sắp xếp mặc định.
Array.Sort(arrayOfCars);
Console.WriteLine("\nArray - Sorted by Company Make (Ascending -
IComparable)\n");

foreach (Car c in arrayOfCars)
    Console.WriteLine(c.Make + "\t\t" + c.Year);

// Sắp xếp tăng dần theo year, sử dụng IComparer.
Array.Sort(arrayOfCars, Car.sortYearAscending());
Console.WriteLine("\nArray - Sorted by Year (Ascending - IComparer)\n");

foreach (Car c in arrayOfCars)
    Console.WriteLine(c.Make + "\t\t" + c.Year);

// Sắp xếp giảm dần theo hãng sản xuất, sử dụng IComparer.
Array.Sort(arrayOfCars, Car.sortMakeDescending());
Console.WriteLine("\nArray - Sorted by Company Make (Descending -
IComparer)\n");

foreach (Car c in arrayOfCars)
    Console.WriteLine(c.Make + "\t\t" + c.Year);

// Sắp xếp giảm dần theo năm sản xuất, sử dụng IComparer.
Array.Sort(arrayOfCars, Car.sortYearDescending());
Console.WriteLine("\nArray - Sorted by Year (Descending - IComparer)\n");

foreach (Car c in arrayOfCars)
    Console.WriteLine(c.Make + "\t\t" + c.Year);

Console.ReadLine();
}

```

Kết quả thực hiện chương trình:

Array - Unsorted

Ford	1992
Fiat	1988
Buick	1932
Ford	1932
Dodge	1999
Honda	1977

Array - Sorted by Company Make (Ascending - IComparable)

Buick	1932
Dodge	1999
Fiat	1988
Ford	1992
Ford	1932
Honda	1977

Array - Sorted by Year (Ascending - IComparer)

Buick	1932
Ford	1932
Honda	1977
Fiat	1988
Ford	1992
Dodge	1999

Array - Sorted by Company Make (Descending - IComparer)

Honda	1977
Ford	1932
Ford	1992
Fiat	1988
Dodge	1999
Buick	1932

Array - Sorted by Year (Descending - IComparer)

Dodge	1999
Ford	1992
Fiat	1988
Honda	1977
Ford	1932
Buick	1932

Ví dụ 2:

```
using System;
using System.Collections;

public class Example
{
    public class ReverserClass : IComparer
    {
        // Viết lại phương thức so sánh hai đối tượng.
        int IComparer.Compare(Object x, Object y)
        {
            return ((new CaseInsensitiveComparer()).Compare(y, x));
        }
    }

    public static void Main()
    {
        // Tạo mảng chuỗi ký tự.
        string[] words = { "The", "quick", "brown", "fox", "jumps", "over",
                           "the", "lazy", "dog" };

        // Hiển thị các giá trị.
        Console.WriteLine("The array initially contains the following values:");
        PrintIndexAndValues(words);

        // Sắp xếp mảng chuỗi ký tự, sử dụng so sánh mặc định.
```

```

        Array.Sort(words);
        Console.WriteLine("After sorting with the default comparer:" );
        PrintIndexAndValues(words);

        // sắp xếp các giá trị mảng bằng cách sử dụng so sánh không phân biệt chữ
        // hoa chữ thường, sử dụng lớp ReverserClass đã viết ở trên.
        Array.Sort(words, new ReverserClass());
        Console.WriteLine("After sorting with the reverse case-insensitive
        comparer:");
        PrintIndexAndValues(words);
    }

    public static void PrintIndexAndValues(IEnumerable list)
    {
        int i = 0;
        foreach (var item in list )
            Console.WriteLine($" [{i++}]: {item}");

        Console.WriteLine();
    }
}

```

Giao diện IComparer với List:

Ví dụ: `List<Student> list = new List<Student>();`

Trong đó lớp `CompareToYear()` được viết bên ngoài class `Student`, thực thi giao diện `Comparer` và viết lại phương thức, ví dụ:

```

class CompareToYear : IComparer<Student>
{
    public int Compare(Student x, Student y)
    {
        return (x.year - y.year);
    }
}

```

Khi cần sắp xếp, sử dụng câu lệnh:

```
list.Sort(new CompareToYear());
```

Tham khảo:

<https://www.c-sharpcorner.com/UploadFile/annathurai/abstract-class-in-C-Sharp/>

<https://www.tutlane.com/tutorial/csharp/csharp-access-modifiers-public-private-protected-internal>

<https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/abstract>

<https://docs.microsoft.com/vi-vn/dotnet/csharp/language-reference/keywords/override>

<https://www.c-sharpcorner.com/article/sealed-class-in-C-Sharp/>

<https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/interface>

<https://zetcode.com/csharp/icomparable/>

<https://docs.microsoft.com/en-us/troubleshoot/dotnet/csharp/use-icomparable-icomparer>

<https://docs.microsoft.com/en-us/dotnet/api/system.collections.icomparer.compare?view=net-5.0>