

LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG

LỜI MỞ ĐẦU

Lập trình theo phương pháp hướng đối tượng xuất hiện từ những năm 1990 và được hầu hết các ngôn ngữ lập trình hiện nay hỗ trợ. Nó có nhiều ưu điểm vượt trội so với phương pháp lập trình cấu trúc cổ điển, nhất là khi phát triển các ứng dụng lớn. Giáo trình này sẽ giới thiệu các đặc trưng của phương pháp lập trình hướng đối tượng như tính đóng gói, tính kế thừa và tính đa hình. Chúng tôi chọn ngôn ngữ C# để minh họa, vì đây là ngôn ngữ lập trình hướng đối tượng dễ học và phổ dụng trong hiện nay trong các lĩnh vực nghiên cứu cũng như trong ngành công nghệ phần mềm. Sau khi hoàn tất giáo trình này, sinh viên sẽ có kỹ năng mô hình hóa các lớp đối tượng trong thế giới thực thành các lớp đối tượng trong C# và cách phối hợp các đối tượng này để giải quyết vấn đề đang quan tâm, như trong lĩnh vực công nghệ phần mềm hay cài đặt các thuật toán học máy, khai phá dữ liệu,...

Trước khi tìm hiểu chi tiết về phương pháp lập trình hướng đối tượng, sinh viên nên đọc trước phần **Phụ lục A - Cơ bản về ngôn ngữ C#** để làm quen với các kiểu dữ liệu, các cấu trúc điều khiển trong ngôn ngữ C#. Sau khi đã nắm bắt được phương pháp lập trình hướng đối tượng, sinh viên nên đọc thêm phần phụ lục **B-Biệt lệ** để có thể viết chương trình có tính dung thứ lỗi cao hơn. Sau mỗi chủ đề trong giáo trình có các bài tập áp dụng kiến thức của chủ đề. Cuối mỗi chủ đề chính còn có bài tập tổng hợp, bài tập tự tìm hiểu. Các bài tập tổng hợp nhằm giúp sinh viên ôn tập lại các kiến thức đã học và rèn luyện kỹ năng phân biệt, phân tích, lý giải các khái niệm đã học.

Chắc chắn sẽ có những điểm thiếu sót hoặc chưa hoàn hảo trong giáo trình này. Chúng tôi rất mong nhận được sự phản hồi, góp ý từ các bạn sinh viên, các đồng nghiệp và quý độc giả để giáo trình hoàn thiện hơn trong lần xuất bản kế tiếp.

Đà Lạt, 10/12/2022

TM. Nhóm tác giả

Chủ biên, Phạm Quang Huy

MỤC LỤC

NỘI DUNG	4
I. Giới thiệu lập trình hướng đối tượng	4
I.1. Lập trình hướng thủ tục (Pascal, C, ...)	4
I.2. Lập trình hướng đối tượng (Object-oriented programming)	4
I.2.1. Tính đóng gói	5
I.2.2. Tính kế thừa	5
I.2.3. Tính đa hình	6
I.2.4. Ưu điểm của phương pháp lập trình hướng đối tượng	6
II. Lớp và đối tượng	7
II.1. Định nghĩa lớp	7
II.2. Tạo đối tượng	9
II.3. Phương thức khởi tạo	Error! Bookmark not defined.
II.4. Phương thức khởi tạo sao chép (copy constructor)	12
II.5. định nghĩa chồng phương thức	13
II.6. Sử dụng các thành viên tĩnh (static)	17
II.7. Tham số của phương thức	Error! Bookmark not defined.
II.7.1. Truyền tham trị bằng tham số kiểu giá trị	Error! Bookmark not defined.
II.7.2. Truyền tham chiếu bằng tham số kiểu giá trị với từ khóa ref	20
II.7.3. Truyền tham chiếu với tham số kiểu giá trị bằng từ khóa out	Error! Bookmark not defined.
II.7.4. Truyền tham trị với tham số thuộc kiểu tham chiếu	21
II.8. Tham chiếu this	24
II.9. Đóng gói dữ liệu với thuộc tính (property)	26
II.10. Toán tử (operator)	29
II.11. Indexer (Chỉ mục)	32
II.12. Lớp lồng nhau	36
II.13. Câu hỏi ôn tập	37
II.14. Bài tập tổng hợp	38
III. Kế thừa (inheritance) và đa hình (polymorphism)	39
III.1. Quan hệ chuyên biệt hóa và tổng quát hóa	39
III.2. Kế thừa	39
III.3. Gọi Phương thức khởi tạo của lớp cơ sở	41
III.4. Định nghĩa phiên bản mới trong lớp dẫn xuất	43
III.5. Tham chiếu thuộc lớp cơ sở	45
III.6. Phương thức ảo (virtual method) và tính đa hình (polymorphism)	47
III.7. Lớp Object	55
III.8. Lớp trừu tượng (abstract)	55
III.9. Giao diện (interface)	58
III.9.1. Thực thi giao diện	58
III.9.2. Hủy đối tượng	60
III.9.3. Thực thi nhiều giao diện	65
III.9.4. Mở rộng giao diện	68
III.9.5. Kết hợp giao diện	68
III.9.6. Kiểm tra đối tượng có hỗ trợ giao diện hay không bằng toán tử <i>is</i> ...	68

III.9.7. Các giao diện Icomparer, IComparable (giao diện so sánh) và ArrayList	69
III.9.8. Câu hỏi ôn tập	78
III.9.9. Bài tập tổng hợp	79
I. Tạo ứng dụng trong C#	80
I.1. Soạn thảo chương trình “Hello World”	81
I.2. Biên dịch và chạy chương trình “Hello World”	82
II. Cơ sở của ngôn ngữ C#	82
II.1. Các kiểu dữ liệu	82
II.1.1. Các kiểu xây dựng sẵn trong C#:	83
II.1.2. Hằng	84
II.1.3. Kiểu liệt kê	84
II.1.4. Kiểu chuỗi	86
II.2. Lệnh rẽ nhánh	86
II.2.1. Lệnh if	86
II.2.2. Lệnh switch	87
II.2.3. Lệnh goto	88
II.2.4. Lệnh lặp while	88
II.2.5. Lệnh do...while	89
II.2.6. Lệnh for	89
II.2.7. Lệnh foreach	90
II.2.8. Lệnh continue và break	91
II.3. Mảng	92
II.3.1. Mảng một chiều	92
II.3.2. Mảng nhiều chiều	93
II.3.3. Một số ví dụ khác về mảng nhiều chiều	94
II.3.4. Truyền tham số cho hàm	95
II.3.5. Truyền tham số cho hàm theo kiểu tham trị	95
II.3.6. Truyền tham số cho hàm theo kiểu tham chiếu	96
II.3.7. Truyền tham số cho hàm với tham số là biến kiểu tham chiếu	97
II.4. Không gian tên (namespace)	98
I. Ném ra biệt lệ	103
II. Bắt ngoại lệ	101
III. Khối finally	103
IV. Một số ngoại lệ khác:	104
V. Một số ví dụ khác	Error! Bookmark not defined.

NỘI DUNG

I. Giới thiệu lập trình hướng đối tượng

1.1. Lập trình hướng thủ tục

Trong phương pháp lập trình thủ tục (procedural programming), chương trình là một hệ thống các thủ tục, hàm được hình thành theo hướng từ trên xuống. Khi lập trình, ta phải xác định chương trình làm những chức năng nào? Mỗi chức năng gồm những thao tác con nào? Chương trình (chức năng) lớn được phân rã thành các khối chức năng con cho đến khi đủ nhỏ để dễ lập trình và kiểm tra. Khi đó, mỗi chức năng, thao tác sẽ tương ứng với một hàm. Như vậy, lập trình theo phương pháp thủ tục là xác định các hàm, định nghĩa các hàm và gọi các hàm này để giải quyết vấn đề được đặt ra.

Việc phân rã, chia chương trình thành các nhóm chức năng, các hàm cho phép nhiều nhóm/người có thể tham gia vào việc xây dựng chương trình. Mỗi người xây dựng một hay một số các hàm độc lập với nhau. Phương pháp này dẫn đến một khái niệm mới – sự trừu tượng hóa theo chức năng để tăng cường tính cộng tác, tính chuyên môn hóa. Tức là, ta chỉ cần biết một hàm nào đó có thể làm được những công việc cụ thể gì mà không cần biết hàm đó được cài đặt như thế nào. Chừng nào hàm còn tin cậy được thì còn có thể dùng nó. Đây là nền tảng của lập trình thủ tục.

Một trong những nhược điểm của phương pháp này là mọi hàm đều có thể truy cập biến toàn cục. Hoặc dữ liệu có thể phải truyền qua rất nhiều hàm trước khi đến được hàm thực sự sử dụng hoặc thao tác trên nó. Ngoài ra, tính modul hóa của phương pháp này chưa cao. Khi chương trình quá lớn thường dẫn đến sự khó kiểm soát, phát triển, sửa đổi và sửa lỗi chương trình.

1.2. Lập trình hướng đối tượng

Phương pháp lập trình hướng đối tượng (Object-oriented programming) lấy đối tượng làm trung tâm để xây dựng chương trình. Ngoài các ưu điểm của phương pháp lập trình thủ tục nó giúp cho việc quản lý mã chương trình thuận tiện (kể cả khi có sự thay đổi), dễ mở rộng dự án, tính an toàn dữ liệu cao, dễ tái sử dụng mã nguồn...

Phương pháp này gom các dữ liệu liên quan đến mỗi loại đối tượng nào đó và các hàm (còn gọi là phương thức) thao tác trên các dữ liệu này vào một đơn vị gọi là lớp (class). Điều này làm tăng tính đóng gói dữ liệu, từ đó làm tăng tính an toàn dữ liệu: các thao tác truy cập đến dữ liệu của một lớp cần phải được thực hiện thông qua các hàm của lớp đó.

$$\textbf{Đối tượng} = \textbf{Dữ liệu} + \textbf{Phương thức}$$

Khi lập trình theo phương pháp hướng đối tượng ta thường phải trả lời các câu hỏi:

- Chương trình (đoạn chương trình) liên quan tới những lớp đối tượng nào?
- Mỗi đối tượng cần có những dữ liệu và thao tác nào?
- Các đối tượng quan hệ với nhau như thế nào trong chương trình?

Từ đó ta thiết kế các lớp đối tượng và tổ chức trao đổi thông tin giữa các đối tượng, ra lệnh để đối tượng thực hiện các nhiệm vụ thích hợp.

Đôi lúc, ta có thể tiếp cận theo cách ngược lại như phương pháp thủ tục: xác định các hàm (thao tác) cần thiết để xử lý các tác vụ nào đó trước, sau đó sắp xếp, tổ chức lại dữ liệu và các hàm thành các lớp đối tượng tương ứng.

Ví dụ:

- Đối tượng tài khoản ngân hàng:
 - Dữ liệu: số tài khoản, mật khẩu, tên chủ sở hữu, số tiền hiện có...
 - Thao tác: đăng nhập, gửi tiền, rút tiền, chuyển khoản...
- Đối tượng chuỗi :
 - Dữ liệu: mảng các kí tự.
 - Thao tác: tính chiều dài, nối hai chuỗi...
- Đối tượng file:
 - Dữ liệu: tên file, đường dẫn, nội dung, các thuộc tính
 - Thao tác: tạo, xóa, sao chép, di chuyển, đổi tên, mở file, sửa nội dung...

Hiện nay nhiều ngôn ngữ lập trình hỗ trợ lập trình hướng đối tượng chẳng hạn như Java, C++, C#, Perl, Python, JavaScript, Simula, Modula, Ada, Smalltalk...

Bốn đặc trưng nổi bật của phương pháp lập trình hướng đối tượng là tính trừu tượng (abstraction), tính đóng gói (encapsulation), tính kế thừa (inheritance) và tính đa hình (polymorphism).

1.2.1. Tính đóng gói

Tính đóng gói thể hiện qua việc ràng buộc dữ liệu và phương thức thao tác trên dữ liệu đó vào trong lớp để dễ kiểm soát, làm tăng tính trừu tượng của dữ liệu. Lớp đối tượng chỉ cung cấp một số phương thức để giao tiếp với môi trường bên ngoài, che dấu đi cài đặt thực sự bên trong của lớp.

Tính đóng gói thể hiện qua việc quy định mức độ truy cập của các thành viên được khai báo, định nghĩa trong lớp như: biến, hàm, thuộc tính, chỉ mục, lớp con...

1.2.2. Tính kế thừa

Tính kế thừa thể hiện qua cơ chế định nghĩa một lớp đối tượng mới dựa trên lớp khác đã có nhằm tận dụng các đoạn mã chương trình đã có. Lớp mới chỉ việc bổ

sung các thành phần riêng của chính nó hoặc định nghĩa lại các hàm của lớp đã có nếu không còn phù hợp với nó.

I.2.3. Tính đa hình

Tính đa hình cho phép định nghĩa một hàm có thể thao xử lý chung một nhóm loại đối tượng mà khi thực hiện hàm thì cách thực hiện có thể khác nhau tùy vào loại đối tượng cụ thể đang được xử lý. Trong nhiều tình huống, điều này làm giảm đáng kể độ phức tạp của chương trình.

Ví dụ, ta có thể cài đặt chung một thuật toán sắp xếp nổi bọt cho các loại dữ liệu số nguyên, số thực, kí tự, chuỗi, hay cho các loại đối tượng có thể so sánh thứ tự với nhau. Trong trường hợp này, việc so sánh thứ tự của hai đối tượng mang tính đa hình vì mỗi loại đối tượng sẽ có cách so sánh phù hợp.

I.2.4. Tính trừu tượng

Có thể xem tính trừu tượng là một phần mở rộng của tính đóng gói. Tính trừu tượng cho phép lập trình viên có thể ẩn/che giấu những cài đặt trong cách xây dựng các lớp đối tượng. Đó là những dữ liệu hoặc các quy trình/cách xử lý mà người sử dụng nên quan tâm hoặc can thiệp để giảm độ phức tạp và tăng hiệu quả phát triển chương trình. Người sử dụng chỉ cần nắm rõ những chức năng và cách thức truyền thông điệp cho các lớp đối tượng đó.

Chẳng hạn, một chiếc xe hoạt động cần rất nhiều bộ phận bên trong làm việc, nhưng tất cả những thông tin đó không cần thiết với người lái. Người lái chỉ cần sử dụng những công cụ như: bàn đạp ga, phanh, vô lăng, đèn nháy,... để biết cách lái xe, chứ không cần quan tâm những chi tiết về cách thức động cơ hoạt động. Tương tự, có rất nhiều cách để máy tính của bạn kết nối với một mạng cục bộ, như Ethernet, Wi-Fi, modem quay số,... Tuy nhiên, trình duyệt Web không phải bận tâm đến việc bạn dùng mạng nào, bởi vì các phần mềm sẽ cung cấp một khái niệm trừu tượng chung để trình duyệt hiểu. Trong trường hợp này, “kết nối mạng” là sự trừu tượng, còn Ethernet và Wi-Fi,... là những triển khai cho sự trừu tượng đó.

I.2.5. Một số ưu điểm của phương pháp lập trình hướng đối tượng

Một số lợi ích của tính trừu tượng có thể kể đến là:

- ✓ Giao diện người dùng đơn giản, súc tích.
- ✓ Các đoạn mã phức tạp bị ẩn đi.
- ✓ Nâng cao vấn đề bảo mật, an toàn dữ liệu. Tính đóng gói làm giới hạn phạm vi sử dụng của các biến, nhờ đó việc quản lý giá trị của biến dễ dàng hơn, việc sử dụng mã an toàn hơn.
- ✓ Việc bảo trì, cập nhật hay thay đổi một mô-đun dễ dàng hơn, ít khi ảnh hưởng tới các mô-đun khác. Phương pháp này làm cho tốc độ phát triển các chương trình nhanh hơn vì mã được tái sử dụng và cải tiến dễ dàng, uyển chuyển. Việc phân module chức năng cũng dễ dàng, thuận tiện hơn.

- ✓ Phương pháp này tiến hành tiến trình phân tích, thiết kế chương trình thông qua việc xây dựng các đối tượng mô các đối tượng thực tế. Điều này làm cho việc sửa đổi dễ dàng hơn khi cần thay đổi chương trình.
- ✓ ...

I.2.6. Một số nhược điểm của phương pháp lập trình hướng đối tượng

So với phương pháp lập trình thủ tục, phương pháp lập trình hướng đối tượng có một vài nhược điểm sau:

- ✓ Khó học hơn so đối với người mới học lập trình.
- ✓ Chương trình có thể chậm hơn một chút nhưng không đáng kể.

II. Lớp và đối tượng

Trong phương pháp lập trình hướng đối tượng, chương trình là một hệ thống các lớp (class), các đối tượng (object). Xây dựng một chương trình là định nghĩa các lớp đối tượng, sau đó tổ chức sắp xếp để các đối tượng phối hợp nhau thực thi nhiệm vụ.

II.1. Định nghĩa lớp

Một lớp là một kiểu cấu trúc mở rộng. Đó là một kiểu mẫu chung cho các đối tượng thuộc cùng một loại. Thành phần của lớp gồm cấu trúc dữ liệu mô tả các đối tượng trong lớp và các phương thức mà mỗi biến đối tượng của lớp đều có. Các phương thức còn gọi là các hàm/thao tác trên các thành phần dữ liệu được khai báo trong lớp.

Các dữ liệu, phương thức được khai báo trong lớp được gọi chung là các thành viên của lớp. Các thành phần dữ liệu được xem như biến toàn cục đối với các phương thức của lớp, tức là các phương thức của lớp có quyền truy cập đến các thành phần dữ liệu này mà không cần phải khai báo lại trong từng phương thức.

Việc định nghĩa lớp thể hiện tính đóng gói và tính trừu tượng của phương pháp lập trình hướng đối tượng.

Cú pháp định nghĩa lớp:

```
[ MứcĐộ Truy Cập ] class TênLớp [ :LớpCơSở ] {
    - Khai báo các thành phần dữ liệu (khai báo biến)
    - Định nghĩa các phương thức, thuộc tính của lớp
}
```

Mức độ truy cập

Mức độ truy cập (**access-modifiers**) được áp dụng cho lớp hoặc một thành viên (biến, phương thức,...) nào đó của lớp. Nó cho biết loại phương thức nào bên ngoài lớp được phép truy cập đến nó, hay nói cách khác nó mô tả phạm vi mà thành phần đó được nhìn thấy bởi các thành phần khác bên ngoài lớp.

Bảng sau liệt kê các kiểu mức độ truy cập của các thành phần trong một lớp:

Mức độ truy cập	Ý nghĩa
public	Thành viên được đánh dấu public được nhìn thấy bởi bất kỳ phương thức nào của lớp khác.
private	Chỉ có các phương thức của lớp A mới được phép truy cập đến thành phần được đánh dấu private trong các lớp A.
protected	Chỉ có các phương thức của lớp A hoặc của lớp dẫn xuất từ A mới được phép truy cập đến thành phần được đánh dấu protected trong lớp A. Mức độ này không áp dụng cho lớp, chỉ áp dụng cho các thành viên.
internal	Các thành viên internal trong lớp A được truy xuất trong các phương thức của bất kỳ lớp trong khối kết hợp (assembly, name space) của A
protected internal	Tương đương với protected or internal

Chú ý:

- ✓ Mặc định, nếu không chỉ cụ thể mức độ truy cập thì thành viên của lớp được xem là có mức độ truy cập private.
- ✓ Mức độ truy cập internal cho phép các phương thức của các lớp trong cùng một khối kết hợp (assembly) với lớp đang định nghĩa có thể truy cập. Các lớp thuộc cùng một project có thể xem là cùng một khối kết hợp.
- ✓ Tìm hiểu thêm về mức độ truy cập tại: <https://openplanning.net/10439/csharp-access-modifiers>

II.2. Tạo đối tượng

Lớp mô tả cấu trúc chung của một nhóm đối tượng nào đó. Một đối tượng là một thể hiện cụ thể của một lớp.

Biến đối tượng là một kiểu tham chiếu (con trỏ). Về bản chất, một biến đối tượng chỉ là một số nguyên dùng để lưu địa chỉ của vùng nhớ chứa dữ liệu thực sự của một đối tượng. Ta phải dùng toán tử **new** để cấp phát vùng nhớ cho đối tượng và có thể gán biến đối tượng trỏ tới vùng nhớ này. Sau khi đối tượng được cấp phát bộ nhớ, ta mới có thể gán các giá trị cho các biến thành viên hay gọi thi hành các phương thức của đối tượng này.

Cú pháp khai báo đối tượng:

TênLớp TênBiếnĐốiTượng;

Cấp phát vùng nhớ cho đối tượng:

TênBiếnĐốiTượng = new TênLớp(DanhSáchĐốiSố);

Cú pháp vừa khai báo vừa cấp phát vùng nhớ cho đối tượng:

TênLớp TênBiếnĐốiTượng = new TênLớp(DanhSáchĐốiSố);

Chú ý:

- ✓ *Sau khi khai báo biến đối tượng thì biến đó chỉ là một con trỏ.*
- ✓ *Sau khi cấp phát bằng từ khóa new thì biến trỏ tới một đối tượng thực sự.*
- ✓ *Khi không cần dùng một biến đối tượng nữa (mất hiệu lực) ta thường gán biến trỏ tới nó bằng null. Bộ dọn rác của trình biên dịch đảm nhiệm việc tự động hủy vùng nhớ mà đối tượng chiếm giữ khi đối tượng đó không còn hiệu lực.*

Ví dụ:

Chương trình nhập chiều dài, chiều rộng của hình chữ nhật và xuất ra diện tích, chu vi của hình chữ nhật.

```
// Định nghĩa lớp đại diện cho 1 hình chữ nhật
class HCN {
    //Các dữ liệu của 1 hình chữ nhật
    protected float Dai, Rong;

    //Các phương thức của 1 hình chữ nhật
    public float ChuVi() {
        return (Dai + Rong) * 2;
    }

    public float DienTich() {
        return Dai * Rong;
    }
}
```

```

public void Nhap()    {
    Console.WriteLine("Nhập chiều dài: ");
    Dai = float.Parse(Console.ReadLine());
    Console.WriteLine("Nhập chiều rộng: ");
    Rong = float.Parse(Console.ReadLine());
}

public void Xuat()    {
    Console.WriteLine("Hình chữ nhật: Dai = {0}, Rong = {1}", Dai, Rong);
}
}

class Application {
    static void Main(string[] args)    {
        HCN h;           // Khai báo biến h thuộc lớp HCN
        h = new HCN();    // Cấp phát vùng nhớ cho 1 hình chữ nhật

        // Gọi thực hiện các phương thức của đối tượng
        h.Nhap();
        h.Xuat();
        Console.WriteLine("Chu vi: {0}", h.ChuVi());
        Console.WriteLine("Diện tích: {0}", h.DienTich());
        Console.ReadLine();
    }
}

```

Trong ví dụ trên, ta định nghĩa một lớp **HCN** là khuôn mẫu của các hình chữ nhật. Mỗi đối tượng thuộc lớp (tức là mỗi hình chữ nhật) này có thành phần dữ liệu là chiều dài và chiều rộng và có các phương thức như: **nhap()**, **xuat()**, **DienTich()**, **ChuVi()**.

Trong hàm **Main()** ta khai báo một đối tượng hình chữ nhật tên là **h**, cấp phát vùng nhớ cho đối tượng này và gọi thực hiện các phương thức của nó.

Nếu ta bỏ đi từ khóa **public** đứng trước mỗi phương thức của lớp **HCN** thì hàm **Main()** sẽ không thể truy cập đến các phương thức của đối tượng **h** vì hàm **Main()** là một hàm bên ngoài lớp **HCN**. Trình biên dịch sẽ báo lỗi vì khi đó các phương thức này có mức độ truy cập là **private**.

Bài tập

Bài tập 1. Xây dựng lớp hình chữ nhật có các cạnh song song với các trục tọa độ. Thành phần dữ liệu của 1 hình chữ nhật gồm tọa độ góc trên bên trái (x_1, y_1) và tọa độ góc dưới bên phải (x_2, y_2). Các phương thức của hình chữ nhật gồm tính chiều dài, chiều rộng, diện tích, chu vi của hình chữ nhật và phương thức vẽ hình chữ nhật bằng các ký tự '*' ra màn hình.

Bài tập 2. Viết chương trình xây dựng lớp phân số và các thao tác trên phân số như +, -, *, /, tìm ước số chung lớn nhất của tử và mẫu, rút gọn, cộng phân số với một số nguyên. Gợi ý:

```

class PhanSo {
    int Tu, Mau; // private members

    public void NhapPhanSo() {
        // Đoạn mã nhập tử số và mẫu số.
    }
}

```

```

public void GanGiaTri(int TuSo, int MauSo)    {
    // Đoạn mã gán giá trị cho tử số và mẫu số.
}

public void XuatPhanSo() {
    // Đoạn mã xuất tử số và mẫu số ở dạng (a/b)
}

// cộng phân số hiện hành với phân số PS2
// và trả về một phân số
public PhanSo Cong(PhanSo PS2) {
    PhanSo KetQua = new PhanSo();
    KetQua.Tu = Tu * PS2.Mau + Mau * PS2.Tu;
    KetQua.Mau = Mau * PS2.Mau;
    return KetQua;
}

// Trừ phân số hiện hành với phân số PS2 và trả về một phân số
public PhanSo Tru(PhanSo PS2){
    ...
}

//... các phương thức khác

```

II.3. Phương thức khởi tạo

Phương thức khởi tạo (constructor) của đối tượng dùng để khởi gán giá trị ban đầu cho các biến dữ liệu của một đối tượng khi đối tượng đó được tạo ra. Nếu ta không định nghĩa một phương thức khởi tạo nào thì trình biên dịch sẽ tự động tạo một phương thức khởi tạo mặc định cho đối tượng thuộc lớp đó và gán giá trị mặc định cho các biến.

Phương thức khởi tạo của một đối tượng có các tính chất sau:

- ✓ Được gọi đến một cách tự động khi một đối tượng của lớp được tạo ra.
- ✓ Tên phương thức giống với tên lớp
- ✓ Thường có mức độ truy cập là public để hàm bên ngoài lớp có thể gọi nó (nhưng không bắt buộc).
- ✓ Không có giá trị trả về.

Thông thường ta nên định nghĩa một phương thức khởi tạo cho lớp và cung cấp tham số cho phương thức khởi tạo để khởi tạo các biến cho đối tượng của lớp.

Chú ý rằng, nếu lớp có phương thức khởi tạo có tham số thì khi khởi tạo đối tượng, bằng toán tử **new**, ta phải truyền tham số cho phương thức khởi tạo theo cú pháp:

TênBiếnĐốiTượng = new TênLớp(DanhSáchĐốiSố);

Ví dụ:

Ví dụ sau xây dựng một lớp **Time** trong đó có một phương thức khởi tạo nhận tham số có kiểu **DateTime** (kiểu xây dựng sẵn của trình biên dịch) làm tham số khởi gán cho các thành phần dữ liệu của đối tượng thuộc lớp **Time**.

```
public class Time    {
    int Year;
    int Month;
    int Date;
    int Hour;
    int Minute;
    int Second = 30;

    public void DisplayTime()    {
        Console.WriteLine("{0}/{1}/{2} {3}:{4}:{5}",
            Month, Date, Year, Hour, Minute, Second);
    }

    // Hàm khởi tạo gán đối tượng Time bằng đối tượng System.DateTime có sẵn
    public Time(System.DateTime dt)    {
        Console.WriteLine("Ham constructor tu dong duoc goi!");
        Year = dt.Year;
        Month = dt.Month;
        Date = dt.Day;
        Hour = dt.Hour;
        Minute = dt.Minute;
    }
}

class DateTimeConstrcutorApp    {
    static void Main()    {
        // Lấy thời gian hiện tại của hệ thống
        DateTime currentTime = DateTime.Now;

        // Hàm khởi tạo đối tượng t bằng thời gian hiện tại của hệ thống
        Time t = new Time(currentTime);
        t.DisplayTime();
        Console.ReadLine();
    }
}
```

Kết quả của chương trình:



Hãy nhấn phím **F11** chạy debug chương trình để hiểu rõ hơn quá trình khởi tạo đối tượng *t*, gọi thực hiện hàm constructor của *t*.

Chú ý rằng, ta cố tình gán giá trị mặc định là 30 cho biến **Second** để biến **Second** của mọi đối tượng thuộc lớp **Time** đều có giá trị là 30 khi mới được tạo ra.

II.4. Phương thức khởi tạo sao chép (copy constructor)

Phương thức khởi tạo sao chép khởi gán giá trị cho đối tượng mới bằng cách sao chép dữ liệu của đối tượng cùng kiểu đang tồn tại. Ví dụ, ta muốn truyền một đối

tượng **Time** **x** để khởi gán cho đối tượng **Time** **y** mới với mục đích làm cho **y** có giá trị giống **x**, ta sẽ xây dựng Phương thức khởi tạo sao chép của lớp **Time** như sau:

```
public Time(Time existingTimeObject) {  
    Year = existingTimeObject.Year;  
    Month = existingTimeObject.Month;  
    Date = existingTimeObject.Date;  
    Hour = existingTimeObject.Hour;  
    Minute = existingTimeObject.Minute;  
    Second = existingTimeObject.Second;  
}
```

Như vậy, khi khởi tạo đối tượng **y** bằng lệnh dạng:

Time y = new Time(x).

thì hàm copy constructor được gọi và gán giá trị của **x** cho **y**.

Bài tập

Bài tập 1: Xây dựng lớp *HocSinh* (họ tên, điểm toán, điểm văn) với các phương thức: khởi tạo, xuất, tính điểm trung bình.

Bài tập 2: Xây dựng lại lớp *PhanSo* phần trước với phương thức khởi tạo gồm 2 tham số.

Bài tập 3: Xây dựng lớp ngăn xếp *Stack* lưu trữ dữ liệu số nguyên bằng mảng với các thao tác cơ bản như: *Push*, *Pop*, kiểm tra tràn stack, kiểm tra stack rỗng... Dữ liệu của một đối tượng thuộc lớp *Stack* gồm: *Data* (mảng số nguyên), *Size* (kích thước của mảng *Data*), *Top* (chỉ số của phần tử nằm trên đỉnh *Stack*).

Bài tập 4: Xây dựng lớp hàng đợi *Queue* lưu trữ dữ liệu số nguyên bằng mảng với các thao tác trên hàng đợi.

II.5. Định nghĩa chồng phương thức

Định nghĩa chồng phương thức (method overloading) là định nghĩa các phương thức/hàm cùng tên nhưng khác tham số hoặc kiểu trả về. Khi chạy chương trình, phương thức thích hợp nhất sẽ được gọi tùy vào tham số được truyền cho phương thức.

Ví dụ 1:

Mình họa việc định nghĩa chồng phương thức khởi tạo để linh động trong cách tạo đối tượng. Lớp **Date** có 3 phương thức khởi tạo có tác dụng lần lượt như sau:

- ✓ **public Date():** khởi tạo đối tượng thuộc lớp **Date** với giá trị mặc định là 1/1/1900.
- ✓ **public Date(int D, int M, int Y):** khởi tạo các giá trị **Day**, **Month**, **Year** của đối tượng thuộc lớp **Date** bằng ba tham số **D**, **M**, **Y**.
- ✓ **public Date(Date ExistingDate):** đây là hàm copy constructor, khởi tạo đối tượng mới thuộc lớp **Date** bằng một đối tượng cùng kiểu đã tồn tại.
- ✓ **public Date(System.DateTime dt):** khởi tạo đối tượng thuộc lớp **Date** bằng dữ liệu của đối tượng thuộc lớp **System.DateTime** (có sẵn).

```
public class Date {
    private int Year;
    private int Month;
    private int Day;

    public void Display() {
        Console.Write("{0}/{1}/{2}", Day, Month, Year);
    }

    // Hàm khởi tạo không tham số: khởi tạo đối tượng bằng 1/1/1900
    public Date() {
        Console.WriteLine("Constructor không tham số!");
        Year = 1900;
        Month = 1;
        Day = 1;
    }

    // Hàm khởi tạo với tham số kiểu DateTime
    public Date(System.DateTime dt) {
        Console.WriteLine("Constructor với tham số kiểu DateTime!");
        Year = dt.Year;
        Month = dt.Month;
        Day = dt.Day;
    }

    // Hàm khởi tạo với 3 tham số kiểu số nguyên
    public Date(int D, int M, int Y) {
        Console.WriteLine("Constructor có 3 tham số!");
        Year = Y;
        Month = M;
        Day = D;
    }

    // Hàm khởi tạo sao chép
    public Date(Date ExistingDate) {
        Console.WriteLine("Copy constructor!");
        Year = ExistingDate.Year;
        Month = ExistingDate.Month;
        Day = ExistingDate.Day;
    }
}

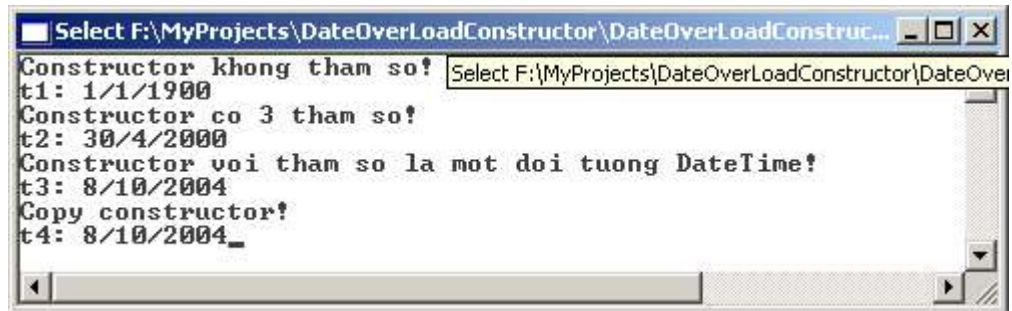
class DateOverLoadConstructorApp {
    static void Main(string[] args) {
        System.DateTime currentTime = System.DateTime.Now;
        Date t1 = new Date(); // t1 = 1/1/1900
        Console.Write("t1: ");
        t1.Display();
        Console.WriteLine();

        Date t2 = new Date(30, 4, 2000);
        Console.Write("t2: ");
        t2.Display();
        Console.WriteLine();

        Date t3 = new Date(currentTime);
        Console.Write("t3: ");
        t3.Display();
        Console.WriteLine();
    }
}
```

```
        Date t4 = new Date(t3);  
        Console.WriteLine("t4: ");  
        t4.Display();  
        Console.ReadLine();  
    }  
}
```

Kết quả của chương trình:



Ví dụ 2:

Định nghĩa chồng phương thức khởi tạo của lớp *PhanSo* để linh động khi tạo ra các đối tượng phân số.

```
using System;  
class PhanSo  
{  
    int Tu, Mau;  
  
    // Hàm khởi tạo gán giá trị mặc định cho Tu và Mau  
    public PhanSo()  
    {  
        Tu = 0;  
        Mau = 1;  
        Console.WriteLine("Constructor 1");  
    }  
  
    // Hàm khởi tạo gán phân số tương đương 1 số nguyên  
    public PhanSo(int x)  
    {  
        Tu = x;  
        Mau = 1;  
        Console.WriteLine("Constructor 2");  
    }  
  
    // Hàm khởi tạo gán phân số với 2 giá trị cho tử và mẫu  
    public PhanSo(int t, int m)  
    {  
        Tu = t;  
        Mau = m;  
        Console.WriteLine("Constructor 3");  
    }  
  
    public void XuatPhanSo()  
    {  
        Console.WriteLine("{0}/{1}", Tu, Mau);  
    }  
}
```

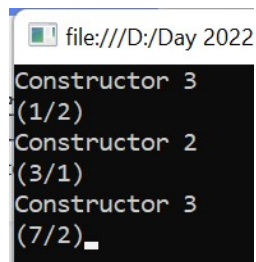
```
public PhanSo Cong(PhanSo PS2)
{
    int TS = Tu * PS2.Mau + Mau * PS2.Tu;
    int MS = Mau * PS2.Mau;

    //Gọi hàm khởi tạo 2 tham số
    PhanSo KetQua = new PhanSo(TS, MS);
    return KetQua;
}
}
class Program
{
    static void Main(string[] args)
    {
        //Gọi hàm khởi tạo 2 tham số
        PhanSo p1 = new PhanSo(1, 2);
        p1.XuatPhanSo();
        Console.WriteLine();

        //Gọi hàm khởi tạo 1 tham số
        PhanSo p2 = new PhanSo(3); // p2 = 3/1
        p2.XuatPhanSo();
        Console.WriteLine();

        PhanSo p3 = p1.Cong(p2);
        p3.XuatPhanSo();
        Console.ReadLine();
    }
}
```

Kết quả chạy chương trình:



```
file:///D:/Day 2022
Constructor 3
(1/2)
Constructor 2
(3/1)
Constructor 3
(7/2)
```

Hãy chạy chương trình ở chế độ dò lỗi (debug) để hiểu rõ khi nào các phương thức được định nghĩa chồng được gọi.

Chú ý rằng, ta có thể định nghĩa phương thức định nghĩa chồng chỉ khác nhau ở từ khóa **ref** hoặc **out** nhưng không thể có hai phương thức chỉ khác nhau ở hai từ khóa **ref** và **out**. Hai từ khóa này sẽ được khảo sát kỹ hơn ở phần sau.

Chẳng hạn, việc định nghĩa chồng như sau là hợp lệ:

```
class MyClass {
    public void MyMethod(int i) {
        i = 10;
    }

    public void MyMethod(ref int i) {
        i = 5;
    }
}
```


nhưng việc định nghĩa chồng như sau là không hợp lệ:

```
class MyClass {
    public void MyMethod(out int i) {
        i = 10;
    }

    public void MyMethod(ref int i) {
        i = 5;
    }
}
```

II.6. Các thành viên tĩnh (static)

Dữ liệu và phương thức của một lớp có thể là thành viên thuộc chung cho tất cả các đối tượng trong lớp đó (còn gọi là thành viên thuộc lớp hay thành viên static và có từ khóa **static** đứng trước) hoặc thuộc riêng từng đối tượng (còn gọi là thành viên thể hiện hay thành viên **non-static** và không có từ khóa **static** đứng trước).

Thành viên thể hiện (**non-static member**) được kết hợp riêng với từng đối tượng của lớp. Khi mỗi đối tượng của lớp được tạo ra, chúng được cấp phát vùng nhớ để chứa các biến thành viên này. Như vậy, trong cùng một lớp, các đối tượng khác nhau có những biến dữ liệu cùng tên, cùng kiểu nhưng được cấp phát ở các vùng nhớ khác nhau và giá trị của chúng cũng có thể khác nhau.

Trong khi đó, thành viên tĩnh (có từ khóa **static** đứng trước) là phần chung của các đối tượng trong cùng một lớp. Tức là, mọi đối tượng thuộc lớp đều dùng chung một bản sao của thành viên tĩnh dù nhiều đối tượng của lớp được tạo ra.

Như vậy, các thành viên thể hiện được xem là toàn cục trong phạm vi từng đối tượng còn thành viên tĩnh được xem là toàn cục trong phạm vi một lớp.

Việc truy cập đến thành viên tĩnh phải thực hiện thông qua tên lớp (không được truy cập thành viên tĩnh thông qua đối tượng) theo cú pháp:

TênLớp.TênThànhViênTĩnh

Chú ý:

- ✓ Thông thường, các hằng hoặc các dữ liệu dùng để tra cứu của lớp thường là các thành viên tĩnh.
- ✓ Phương thức tĩnh thao tác thường trên các dữ liệu tĩnh.
- ✓ Phương thức tĩnh khi truy cập đến các thành viên không tĩnh phải thông qua một đối tượng, theo cú pháp:

TênĐốiTượng.TênThànhViênThểHiện

II.7. Phương thức khởi tạo tĩnh

Ta có thể định nghĩa một phương thức khởi tạo tĩnh. Phương thức này chạy đầu tiên và chỉ một lần khi ta truy cập đến một phần tử nào đó của lớp hoặc đối tượng. Do vậy phương thức này dùng để khởi gán giá trị cho biến tĩnh của lớp và/hoặc thực hiện một số công việc chuẩn bị nào đó.

Ví dụ: Biến thành viên tĩnh được dùng với mục đích theo dõi số thể hiện hiện tại của lớp.

```
public class Cat    {
    private static int SoMeo = - 6; // biến tĩnh
    private string TenMeo ;

    // Phương thức tạo lập của đối tượng
    public Cat( string T) {
        TenMeo = T ;
        Console.WriteLine("WOAW!!!! {0} day!", TenMeo);
        SoMeo++;
    }

    // Phương thức tạo lập tĩnh (không có mức độ truy cập      public, private...) ,
    // được chạy đầu tiên
    static Cat( )    {
        Console.WriteLine("Bat dau lam thit meo !!!!");
        SoMeo = 0;
    }

    public static void HowManyCats( ){
        Console.WriteLine("Dang lam thit {0} con      meo!",SoMeo);
    }
}

public class Tester {
    static void Main( ) {
        // Phương thức tạo lập tĩnh được gọi thực hiện đầu tiên,
        // trước khi thực hiện hàm HowManyCats( )
        Cat.HowManyCats( );
        Cat tom = new Cat("Meo Tom");
        Cat.HowManyCats( );
        Cat muop = new Cat("Meo Muop");
        Cat.HowManyCats( );
        //tom.HowManyCats( ); ----> Error!
        Console.ReadLine();
    }
}
```

Trong ví dụ này, ta xây dựng lớp **Cat** với một biến tĩnh **SoMeo** để đếm số thể hiện (số mèo) hiện có và một biến thể hiện **TenMeo** để lưu tên của từng đối tượng mèo. Như vậy, mỗi đối tượng **tom**, **muop** đều có riêng biến **TenMeo** và chúng dùng chung biến **SoMeo**.

Ban đầu biến **SoMeo** được khởi gán giá trị -6, nhưng khi gọi lệnh **Cat.HowManyCats()** thì Phương thức khởi tạo tĩnh **static Cat()** tự động thực hiện trước và gán lại giá trị 0 cho biến tĩnh này. Điều này đảm bảo trước khi tạo ra đối tượng Cat đầu tiên thì biến **SoMeo** luôn = 0.

Mỗi khi một đối tượng thuộc lớp **Cat** được tạo ra thì Phương thức khởi tạo của đối tượng này truy cập đến biến đếm **SoMeo** và tăng giá trị của biến này lên một đơn vị. Như vậy, khi đối tượng **tom** được tạo ra, giá trị của biến này tăng lên thành 1, khi đối tượng **muop** được tạo ra, giá trị của biến này tăng lên thành 2.

Phương thức tĩnh **HowManyCats()** thực hiện nhiệm vụ xuất biến tĩnh **SoMeo** thông qua tên lớp bằng câu lệnh:

```
Cat.HowManyCats( );
```

Nếu ta gọi lệnh sau thì trình biên dịch sẽ báo lỗi:

```
tom.HowManyCats( );
```

Kết quả chạy chương trình:



Bài tập: Xây dựng lớp *MyDate* lưu trữ các giá trị ngày, tháng, năm với các phương thức: constructor với 3 tham số, xuất, kiểm tra năm nhuận, tính số ngày của tháng theo tháng và năm, xác định ngày kế tiếp của đối tượng ngày/ tháng/ năm hiện hành.

Gợi ý: để tính số ngày của tháng ta có thể dùng một mảng tĩnh {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31} để lưu số ngày tương ứng với từng tháng. Tuy nhiên với tháng 2 thì tùy năm có nhuận hay không mà ta tính ra giá trị tương ứng.

II.8. Mảng các đối tượng

II.9. Tham số của phương thức

Trong C#, ta có thể truyền tham số cho phương thức (hàm) theo kiểu tham chiếu hoặc tham trị. Nếu muốn truyền tham số cho phương thức theo kiểu tham chiếu thì trước tham số phải có từ khóa **ref** hoặc **out**. Biến tham chiếu **ref** dùng để đưa tham số vào trong phương thức nên chúng phải khác **null** trước khi gọi hàm. Biến này cũng có thể dùng để nhận giá trị trả về của hàm. Biến tham chiếu **out** dùng để nhận giá trị trả về của hàm khi hàm có nhiều giá trị cần trả về nên chúng phải khác **null** trước khi hàm kết thúc.

Điều quan trọng hơn cần nhớ về các kiểu truyền tham số cho phương thức:

- 1) Khi truyền tham số kiểu tham trị thì biến tham (tức là biến tham số hình thức khi định nghĩa hàm) và biến đối số (tức là biến tham số thực khi gọi hàm) là hai biến khác nhau nhưng có cùng giá trị, do có sự sao chép giá trị từ đối số sang biến tham số. Khi đó, một trong có hai trường hợp sau xảy ra:
 - ✓ Nếu biến tham số thuộc kiểu dữ liệu giá trị như **int**, **long**, **float**, **char**,... thì biến tham số hoàn toàn độc lập vùng nhớ với đối số. Như vậy, mọi câu lệnh bên trong hàm chỉ làm thay đổi tham số nhưng không ảnh hưởng đến đối số.

- ✓ Nếu biến tham số thuộc kiểu dữ liệu tham chiếu như đối tượng hay mảng thì chúng là hai biến khác nhau nhưng đều trỏ tới cùng một vùng nhớ. Như vậy, mọi câu lệnh bên trong hàm là thay đổi vùng nhớ dùng chung này thông qua tham số sẽ ảnh hưởng đến cả đối số. Nhưng, mọi câu lệnh bên trong hàm làm tham số trỏ tới vùng nhớ khác sẽ không ảnh hưởng đến đối số. Tức là, nếu đối số trỏ tới đối tượng nào thì ta chỉ có thể làm thay đổi các giá trị dữ liệu của đối tượng đó nhưng không thể làm đối số trỏ tới đối tượng khác.
- 2) Khi truyền tham số kiểu tham chiếu (có từ khóa **ref** hoặc **out**) thì biến tham số chính là đối số đang truyền cho hàm. Trường hợp này có thể xem như một biến có hai tên. Như vậy, mọi câu lệnh bên trong hàm là thay đổi tham số cũng sẽ ảnh hưởng đến đối số.

Các trường hợp sau đây minh họa các chú ý trên.

II.9.1. Hàm trả về nhiều giá trị thông qua tham số là tham chiếu

Một phương thức/hàm chỉ có thể trả về một giá trị bằng lệnh **return**. Nếu muốn phương thức trả về nhiều giá trị, ta có thể dùng cách thức truyền tham chiếu. (Hoặc dùng một cấu trúc/lớp có thể chứa các giá trị trả về.) Ví dụ, phương thức *GetTime* trong lớp *Time* sau đây trả về các giá trị *Hour*, *Minute*, *Second*.

```
public class Time {
    private int Hour;
    private int Minute;
    private int Second;

    public void Display( ) {
        Console.WriteLine("{0}:{1}:{2}", Hour, Minute, Second);
    }

    public Time(System.DateTime dt) {
        Hour = dt.Hour;
        Minute = dt.Minute;
        Second = dt.Second;
    }

    public void GetTime(ref int h, ref int m, ref int s){
        h = Hour;
        m = Minute;
        s = Second;
    }
}

public class PassingParameterByRef {
    static void Main( ) {
        DateTime currentTime = DateTime.Now;
        Time t = new Time(currentTime);
        t.Display( );

        // Trước khi truyền tham chiếu ref cho hàm GetTime các tham chiếu
        // này phải khác null.
        int theHour = 0;
        int theMinute = 0;
        int theSecond = 0;
    }
}
```

```
t.GetTime(ref theHour, ref theMinute, ref theSecond);
Console.WriteLine("Current time: {0}:{1}:{2}", theHour, theMinute,
theSecond);
Console.ReadLine();
}
}
```

C# quy định tất các biến phải được gán giá trị trước khi sử dụng. Vì vậy, trong ví dụ trên, nếu chúng ta không khởi tạo các biến *theHour*, *theMinute* bằng giá trị 0 thì trình biên dịch sẽ thông báo lỗi.

Từ khóa *out* cho phép ta sử dụng tham chiếu mà không cần phải khởi gán giá trị đầu. Trong ví dụ trên, ta có thể sửa phương thức *GetTime* thành:

```
public void GetTime(out int h, out int m, out int s)
```

và hàm *Main()* được sửa lại như sau:

```
static void Main( ) {
    DateTime currentTime = DateTime.Now;
    Time t = new Time(currentTime);
    t.Display( );
    /*int theHour = 0;
    int theMinute = 0;
    int theSecond = 0;*/
    int theHour;
    int theMinute;
    int theSecond;
    t.GetTime(out theHour, out theMinute, out theSecond);
    Console.WriteLine("Current time: {0}:{1}:{2}", theHour, theMinute,
theSecond);
    Console.ReadLine();
}
```

II.9.2. Truyền tham số cho hàm với biến tham số là đối tượng

Nhắc lại:

- ✓ Khi biến tham số của một hàm là một đối tượng và được truyền theo kiểu tham trị (không có từ khóa *ref* hoặc *out*) thì biến tham số và biến đối số đang truyền cho hàm là hai biến khác nhau nhưng cùng trỏ tới một đối tượng. Do vậy, ta chỉ có thể thực hiện các thao tác làm thay đổi các dữ liệu thành phần của đối tượng (thông qua tham số) nhưng các thao tác làm tham số trỏ tới đối tượng nhớ khác không có tác dụng với đối số.
- ✓ Khi một đối tượng được truyền cho hàm theo kiểu tham chiếu (có từ khóa *ref* hoặc *out*) thì biến tham số chính là biến đối số đang truyền cho hàm (một biến có hai tên và đang trỏ tới một đối tượng). Như vậy, mọi câu lệnh bên trong hàm là thay đổi tham số cũng sẽ ảnh hưởng đến đối số.

Ví dụ 1:

Trong lớp **PhanSo** sau đây, các hàm **NghichDao1(PhanSo p)** và **NghichDao3(ref PhanSo p)** có tác dụng nghịch đảo phân số, trong khi hàm **NghichDao2(PhanSo p)** không có tác dụng.

```
using System;
class PhanSo
{
    int Tu, Mau;

    public PhanSo(int x)
    {
        Tu = x;
        Mau = 1;
    }

    public PhanSo(int t, int m)
    {
        Tu = t;
        Mau = m;
    }

    public void XuatPhanSo()
    {
        Console.Write("{0}/{1}", Tu, Mau);
    }

    //Hàm này có tác dụng nghịch đảo vì nó làm thay đổi các biến dữ liệu của p
    public static void NghichDao1(PhanSo p)
    {
        int t = p.Tu;
        p.Tu = p.Mau;
        p.Mau = t;
    }

    // Hàm này không có tác dụng nghịch đảo đối số khi gọi hàm
    // lệnh thay đổi vùng nhớ của tham trị p (p = p2;) chỉ có tác dụng bên
    trong hàm.
    public static void NghichDao2(PhanSo p)
    {
        PhanSo p2 = new PhanSo(0);    // p2 = 0/1
        p2.Mau = p.Tu;
        p2.Tu = p.Mau;
        p = p2;
    }

    //Hàm này có tác dụng nghịch đảo vì có từ khóa ref nên p cũng là đối số
    // lệnh thay đổi vùng nhớ của tham trị p (p = p2;) có tác dụng với đối số
    public static void NghichDao3(ref PhanSo p)
    {
        PhanSo p2 = new PhanSo(0);
        p2.Mau = p.Tu;
        p2.Tu = p.Mau;
        p = p2;
    }
}

class Program
{
```

```
static void Main(string[] args)
{
    Console.Write("Phân số x = ");
    PhanSo x = new PhanSo(3);           // x = 3/1
    x.XuatPhanSo();                      // 3/1

    Console.WriteLine("\nNghịch Đảo 1 có tác dụng với x! x = ");
    PhanSo.NghichDao1(x);
    x.XuatPhanSo();                      // 1/3

    PhanSo y = new PhanSo(2, 5);        // y = 2/5
    Console.WriteLine("\nPhân số y = ");
    y.XuatPhanSo();                      // 2/5
    Console.WriteLine("\nNghịch Đảo 2 không có tác dụng với y! y = ");
    PhanSo.NghichDao2(y);
    y.XuatPhanSo();                      // vẫn là 2/5

    Console.WriteLine("\nNghịch Đảo 3 có tác dụng với y! y = ");
    PhanSo.NghichDao3(ref y);
    y.XuatPhanSo();                      // 5/2
    Console.ReadLine();
}
}
```

Giải thích:

Phân số p là đối tượng được truyền cho hàm **NghichDao1(PhanSo p)** theo kiểu tham trị. Trong hàm **Main()**, khi gọi lệnh **PhanSo.NghichDao1(x);** thì biến tham số p và biến đối số x là 2 biến khác nhau nhưng có cùng giá trị, tức là cùng trỏ tới một vùng nhớ lưu trữ phân số $3/1$. Các thao tác làm hoán vị biến T_u và biến Mau của vùng nhớ này thông qua tham số p cùng tương ứng với việc hoán vị hai biến này thông qua đối số x . Vì vậy, hàm **NghichDao1(PhanSo p)** có tác dụng.

Phân số p là đối tượng cũng được truyền cho hàm **NghichDao2(PhanSo p)** theo kiểu tham trị. Trong hàm **Main()**, khi gọi lệnh **PhanSo.NghichDao2(y);** thì biến tham số p và biến đối số y là 2 biến khác nhau nhưng có cùng giá trị, tức là cùng trỏ tới một vùng nhớ lưu trữ phân số $2/5$. Trong hàm này, ta tạo ra một phân số mới, p_2 có giá trị bằng $5/2$, là nghịch đảo của đối tượng phân số do p và y đang trỏ tới. Tuy nhiên, lệnh gán $p = p_2$ chỉ làm biến p trỏ tới phân số mới ($5/2$) trong khi y vẫn như cũ. Vì vậy, hàm **NghichDao2(PhanSo p)** không có tác dụng.

Phân số p là đối tượng được truyền cho hàm **NghichDao3(ref PhanSo p)** theo kiểu tham chiếu. Trong hàm **Main()**, khi gọi lệnh **PhanSo.NghichDao3(y);** thì biến tham số p cũng chính là biến đối số y . Trong hàm này, ta tạo ra một phân số mới, p_2 có giá trị bằng $5/2$, là nghịch đảo của đối tượng phân số do p (cũng chính là y) đang trỏ tới. Lệnh gán $p = p_2$ tương đương với lệnh gán $y = p_2$ và làm biến y trỏ tới phân số mới ($5/2$). Vì vậy, hàm **NghichDao3(ref PhanSo p)** có tác dụng.

Bài tập 1. Hãy viết phương thức/hàm để hoán vị hai dữ liệu kiểu chuỗi (string). Hãy giải thích tại sao khi dùng phương thức này ta cần phải truyền tham số theo kiểu tham chiếu cho phương thức dù string là dữ liệu thuộc kiểu tham chiếu.

Bài tập 2. Viết chương trình nhập một lớp gồm N học sinh, mỗi học sinh có các thông tin như: họ, tên, điểm văn, điểm toán, điểm trung bình.

Tính điểm trung bình của từng học sinh theo công thức: $(\text{điểm văn} + \text{điểm toán})/2$.

- Tính trung bình điểm văn của cả lớp.
- Tính trung bình điểm toán của cả lớp.
- Sắp xếp học sinh trong lớp theo thứ tự họ tên.
- Sắp xếp học sinh trong lớp theo thứ tự không giảm của điểm trung bình, nếu điểm trung bình bằng nhau thì sắp xếp theo tên.

II.10. Tham chiếu *this*

Trong C#, tại mỗi thời điểm chỉ có một hàm được thực thi. Mỗi khi gọi thực thi một phương thức của một đối tượng (không phải là phương thức tĩnh) thì tham chiếu **this** tự động trỏ đến đối tượng này. Mọi phương thức của đối tượng đều có thể tham chiếu đến các thành phần khác của đối tượng thông qua tham chiếu **this**. Có 3 trường hợp thường dùng tham chiếu **this**:

- ✓ Tránh xung đột tên khi tham số của phương thức trùng tên với tên biến dữ liệu của đối tượng.
- ✓ Dùng để truyền đối tượng hiện tại làm tham số cho một phương thức khác (chẳng hạn gọi đệ qui)
- ✓ Dùng với mục đích chỉ mục.

Ví dụ 1: Dùng tham chiếu **this** với mục đích tránh xung đột tên của tham số với tên biến dữ liệu của đối tượng.

```
public class Date {
    private int Year;
    private int Month;
    private int Day;

    public Date(int Day, int Month, int Year) {
        Console.WriteLine("Constructor có 3 tham số!");
        this.Year = Year;
        this.Month = Month;
        this.Day = Day;
    }
    //... các phương thức khác
}
```

Ví dụ 2: bài toán tháp Hà Nội, minh họa dùng tham chiếu **this** trong đệ qui.

```
class Cot {
    char Ten;
    uint[] Disks;
    static uint Size = 10;
    uint Top = 0;

    public Cot(char TenCot) {
        Disks = new uint[Size];
        Top = 0;
        Ten = TenCot;
    }
}
```



```

//Tạo cột A có n đĩa, đĩa lớn nằm dưới đĩa nhỏ
public Cot(char TenCot, uint SoDia) {
    Disks = new uint[Size];
    Top = SoDia;
    Ten = TenCot;
    for (uint n = SoDia; n > 0; n--) Disks[SoDia - n] = n;
}

//Chuyển đĩa ở đỉnh của cột hiện tại sang cột Dich
public void chuyen1Dia(Cot Dich) {
    this.Top--;
    Console.WriteLine("Chuyen dia {0} tu {1} sang {2} ",
        this.Disks[Top], this.Ten, Dich.Ten);
    Dich.Disks[Dich.Top] = this.Disks[Top];
    Dich.Top++;
}

//Chuyển n đĩa trên cùng từ cột hiện tại sang cột C, lấy B làm trung gian
public void chuyenNhiềuDia(uint n, Cot B, Cot C) {
    if (n == 0) return;
    if (n == 1) chuyen1Dia(C);
    else {
        //Chuyển n -1 đĩa từ cột hiện tại sang cột C, cột B làm trung gian
        this.chuyenNhiềuDia(n - 1, C, B);

        //Chuyển đĩa ở đỉnh của cột hiện tại sang cột C
        this.chuyen1Dia(C);

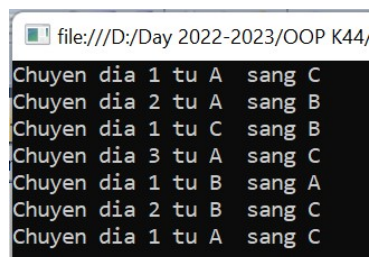
        //Chuyển n -1 đĩa trên cùng từ cột B sang cột C
        //lấy cột hiện tại làm trung gian
        //Dùng tham chiếu this để truyền tham số cho lệnh gọi đệ quy
        B.chuyenNhiềuDia(n-1, this, C);
    }
}

}

class Program {
    static void Main(string[] args) {
        //Tạo cột A có 3 đĩa, cột B, C trống
        Cot A = new Cot('A', 3);
        Cot B = new Cot('B');
        Cot C = new Cot('C');
        //Chuyển n đĩa từ cột A sang cột C, lấy B làm trung gian
        A.chuyenNhiềuDia(3, B, C);
        Console.ReadLine();
    }
}

```

Kết quả chạy chương trình:



```

file:///D:/Day 2022-2023/OOP K44/
Chuyen dia 1 tu A sang C
Chuyen dia 2 tu A sang B
Chuyen dia 1 tu C sang B
Chuyen dia 3 tu A sang C
Chuyen dia 1 tu B sang A
Chuyen dia 2 tu B sang C
Chuyen dia 1 tu A sang C

```

II.11. Đóng gói dữ liệu với thuộc tính (property)

Thuộc tính là một đặc tính mới được giới thiệu trong ngôn ngữ C# làm tăng sức mạnh của tính đóng gói.

Một thuộc tính thường quản lý một biến dữ liệu của đối tượng hoặc của lớp với phương thức **get** để lấy giá trị của biến hoặc phương thức **set** để gán giá trị cho biến đó, hoặc cả hai. Như vậy, thuộc tính cho phép truy cập đến các thành phần dữ liệu của đối tượng/lớp ở mức độ đọc hoặc ghi hoặc cả hai và che dấu cài đặt thực sự bên trong lớp.

Những phương thức/hàm tính toán một đại lượng nào đó của đối tượng mà không cần thêm tham số bên ngoài thì nên định nghĩa thành thuộc tính chỉ có phương thức **get** để mã chương trình đẹp hơn. Chẳng hạn, phương thức/thuộc tính tính điểm trung bình của một học sinh. Trong trường hợp này, thuộc tính tương đương với một hàm không tham số.

Tóm lại, thuộc tính có một trong ba tính chất sau:

- ❖ Chỉ đọc (*read-only*): chỉ có phương thức **get**. Ta chỉ được đọc giá trị của thuộc tính.
- ❖ Chỉ ghi (*write-only*): chỉ có phương thức **set**. Ta chỉ được gán (ghi dữ liệu) giá trị cho thuộc tính.
- ❖ Vừa đọc vừa ghi (*read/write*): có cả hai phương thức **get** và **set**. Được phép đọc và ghi giá trị.

Cú pháp định nghĩa một thuộc tính:

```
public KiểuTrảVề TênThuộcTính {  
    // phương thức lấy giá trị của thuộc tính  
    get {  
        //...các câu lệnh  
        return BiểuThứcTrảVề;  
    }  
  
    //phương thức gán giá trị cho thuộc tính (biến)  
    set {  
        //...các câu lệnh  
        BiếnThànhViên = value;  
    }  
}
```

Ta thấy rằng, cú pháp định nghĩa một thuộc tính khá giống cú pháp định nghĩa một hàm không có tham số. Phần thân của phương thức **get** của thuộc tính tương tự như phần thân phương thức của lớp. Chúng trả về một giá trị hoặc một biến nào đó, thường là trả về giá trị của biến thành viên mà thuộc tính đó quản lý. Khi ta truy xuất đến thuộc tính để lấy giá trị thì phương thức **get** được gọi thực hiện.

Phương thức **set** của thuộc tính dùng để gán giá trị cho biến thành viên mà thuộc tính quản lý. Khi định nghĩa phương thức **set**, ta phải sử dụng từ khóa **value** để biểu diễn cho giá trị dùng để gán cho biến thành viên. Khi gán một giá trị cho thuộc tính thì phương thức **set** tự động được gọi và tham số ẩn **value** chính là giá trị dùng để gán.

Ví dụ: Định nghĩa lớp Student với 3 thuộc tính tương ứng với điểm toán, điểm tin và

```
using System;

class Student {
    //Chú ý rằng tên của các biến có dấu "_" còn tên của      //các thuộc
    tính tương ứng thì không có dấu "_".
    string _Ten ;
    float _DiemToan, _DiemTin;
    float _DiemTB;

    // constructor
    public Student() {
        _Ten = "";
        _DiemToan = 0;
        _DiemTin = 0;
        _DiemTB = 0;
    }

    // thuộc tính ten - (read/write)
    public string Ten {
        get { return _Ten; }
        set { _Ten = value; }
    }

    //Thuộc tính diem toan - (read/write)
    public float DiemToan {
        get { return _DiemToan; }
        set {
            _DiemToan = value;
            _DiemTB = (_DiemToan + DiemTin)/2;
        }
    }

    //Thuộc tính diem tin - (read/write)
    public float DiemTin {
        get { return _DiemTin; }
        set {
            _DiemTin = value;
            _DiemTB = (_DiemToan + DiemTin)/2;
        }
    }

    //Thuộc tính diem tin - (read only)
    public float DiemTrungBinh{
        get { return _DiemTB; }
    }
}
```

```
class Student_PropertyApp {
    static void Main(string[] args) {
        Student s1 = new Student();
        s1.Ten = "Hoa";
        s1.DiemToan = 5;
        s1.DiemTin = 7;
        //s1.DiemTrungBinh = 6; ---> loi
        Console.WriteLine("Ten: {0}, diem Toan: {1}, diem Tin: {2}, diem
        trung binh: {3}", s1.Ten, s1.DiemToan, s1.DiemTin,
        s1.DiemTrungBinh);
        Console.ReadLine();
    }
}
```

Trong ví dụ trên, **Ten**, **DiemToan**, **DiemTin** là các thuộc tính vừa đọc vừa ghi và **DiemTrungBinh** là thuộc tính chỉ đọc. Chúng phủ lên các thành phần dữ liệu tương ứng là **_Ten**, **_DiemToan**, **_DiemTin**, **_DiemTB**, giúp người thiết kế che dấu cài đặt thực sự bên trong lớp. Các thuộc tính **DiemToan**, **DiemTin**, **DiemTrungBinh** không cho phép người dùng gán giá trị cho biến **_DiemTB** và che dấu cách cài đặt của điểm trung bình.

Khi ta gọi các lệnh:

```
s1.Ten = "Hoa";
s1.DiemToan = 5;
s1.DiemTin = 7;
```

thì phương thức **set** của các thuộc tính **Ten**, **DiemToan**, **DiemTin** tương ứng được gọi và tham số ẩn **value** lần lượt là **"Hoa"**, **5**, **7**.

Khi ta gọi các lệnh:

```
Console.Write("Ten: {0}, diem Toan: {1}, ", s1.Ten, s1.DiemToan);
Console.WriteLine(" diem Tin: {0}, diem TB: {1}", s1.DiemTin,
s1.DiemTrungBinh);
```

thì phương thức **get** của các thuộc tính **Ten**, **DiemToan**, **DiemTin**, **DiemTrungBinh** tương ứng được gọi.

Nếu ta gọi lệnh:

```
s1.DiemTrungBinh = 6;
```

thì trình biên dịch sẽ báo lỗi vì thuộc tính **DiemTrungBinh** không cài đặt phương thức **set**.

Bài tập 1: Viết chương trình xây dựng lớp *TamGiac* với dữ liệu là 3 cạnh của tam giác. Xây dựng các thuộc tính (property) *ChuVi*, *DienTich* và các phương thức kiểm tra kiểu của tam giác (thường, vuông, cân, vuông cân, đều).

Bài tập 2: Viết chương trình xây dựng lớp *HinhTruTron* (hình trụ tròn) với dữ liệu chiều cao và bán kính. Xây dựng các thuộc tính (property) *DienTichDay* (diện tích mặt đáy), *DienTichXungQuanh* (diện tích mặt xung quanh), *TheTich* (thể tích).

II.12. Toán tử (operator)

Trong C#, toán tử là một phương thức tĩnh (**static**) dùng để định nghĩa chồng một phép toán nào đó trên các đối tượng. Mục đích của toán tử là để viết mã chương trình gọn gàng, dễ hiểu hơn, thay vì phải gọi phương thức.

Ta có thể định nghĩa chồng các toán tử sau:

- ✓ Toán học: +, -, *, /, %.
- ✓ Cộng trừ một ngôi: ++, --, -.
- ✓ Quan hệ so sánh: ==, !=, >, <, >=, <=.
- ✓ Ép kiểu: ().
- ✓ ...

Cú pháp khai báo nguyên mẫu của một toán tử T:

```
public static KiểuTrảVề operator T (CácThamSố){
    ///các câu lệnh trong thân toán tử
}
```

Ví dụ, toán tử cộng 2 phân số được định nghĩa như sau:

```
// toán tử cộng hai phân số
public static PhanSo operator +(PhanSo PS1, PhanSo PS2) {
    int MauMoi = PS1.Mau * PS2.Mau;
    int TuMoi = PS2.Mau * PS1.Tu + PS1.Mau * PS2.Tu;
    return new PhanSo(TuMoi, MauMoi);
}
```

Một số chú ý:

- ❖ Tham số của toán tử phải là kiểu tham trị (không dùng các từ khóa *ref*, *out*).
- ❖ Không được định nghĩa chồng toán tử = (gán), &&, || (and, or logic), []?: (điều kiện), checked, unchecked, new, typeof, as, is.
- ❖ Khi định nghĩa chồng các toán tử dạng: +, -, *, /, % thì các toán tử +=, -=, *=, /=, %= cũng tự động được định nghĩa chồng.
- ❖ Khi định nghĩa chồng toán tử thì nên định nghĩa chồng theo cặp đối ngẫu. Chẳng hạn, khi định nghĩa chồng toán tử == thì định nghĩa chồng thêm toán tử !=.
- ❖ Khi định nghĩa chồng toán tử ==, != thì nên định nghĩa thêm các phương thức *Equals()*, *GetHashCode()* để đảm bảo quy luật “hai đối tượng bằng nhau theo toán tử == hoặc phương thức *Equals* sẽ có cùng mã băm”.
- ❖ Khi định nghĩa toán tử ép kiểu ta phải chỉ ra đây là toán tử ép kiểu ngầm định (implicit) hay tường minh (explicit).

Cú pháp định nghĩa toán tử ép kiểu:

```
public static [ implicit | explicit ] operator KiểuTrảVềT (Type V){
```

//return 1 biểu thức liên quan tới V

}

Ý nghĩa: Chuyển đổi dữ liệu V kiểu *Type* sang dữ liệu kiểu *Kiểu Trả Về T*.

Ví dụ: xây dựng lớp phân số và định nghĩa chồng các phép toán trên phân số.

```
class PhanSo
{
    int Tu, Mau;    // private members

    //constructor
    public PhanSo(int TuSo, int MauSo)    {
        Tu = TuSo;
        Mau = MauSo;
    }

    //constructor
    public PhanSo(int HoleNumber)    {
        Tu = HoleNumber;
        Mau = 1;
    }

    //constructor
    public PhanSo()    {
        Tu = 0;
        Mau = 1;
    }

    // Ép kiểu ngầm định (tự động) từ số nguyên sang phân số
    public static implicit operator PhanSo(int theInt)    {
        Console.WriteLine("Chuyen so nguyen thanh phan so");
        return new PhanSo(theInt);
    }

    // Ép kiểu tường minh (chủ động) từ số nguyên sang phân số
    public static explicit operator int(PhanSo PS)    {
        return PS.Tu / PS.Mau;
    }

    // so sanh ==
    public static bool operator ==(PhanSo PS1, PhanSo PS2)    {
        return (PS1.Tu * PS2.Mau == PS2.Tu * PS1.Mau);
    }

    // so sanh !=;
    public static bool operator !=(PhanSo PS1, PhanSo PS2)    {
        return !(PS1 == PS2);
    }

    // so sanh 2 phan so co bang nhau hay khong
    public override bool Equals(object o)    {
        Console.WriteLine("Phuong thuc Equals");
        if (!(o is PhanSo)) return false;
        return this == (PhanSo)o;
    }

    // toán tử cộng hai phân số
```

```

public static PhanSo operator +(PhanSo PS1, PhanSo PS2)    {
    int MauMoi = PS1.Mau * PS2.Mau;
    int TuMoi = PS2.Mau * PS1.Tu + PS1.Mau * PS2.Tu;
    return new PhanSo(TuMoi, MauMoi);
}

// Tang phan so them mot don vi!
public static PhanSo operator ++(PhanSo PS)    {
    PS.Tu = PS.Mau + PS.Tu;
    return PS;
}

// Ép phân số về true nếu mẫu số khác 0, ngược lại ép về false
public static implicit operator bool(PhanSo PS)    {
    return PS.Mau != 0;
}

// Hàm đổi số thành chuỗi
public override string ToString()    {
    return Tu.ToString() + "/" + Mau.ToString();
}
}
class PhanSoApp{
    static void Main()    {
        PhanSo f1 = new PhanSo(3, 4);
        Console.WriteLine("f1: {0}", f1.ToString());

        PhanSo f2 = new PhanSo(2, 4);
        Console.WriteLine("f2: {0}", f2.ToString());

        // gọi toán tử cộng hai phân số
        PhanSo f3 = f1 + f2;
        Console.WriteLine("f1 + f2 = f3: {0}", f3.ToString());

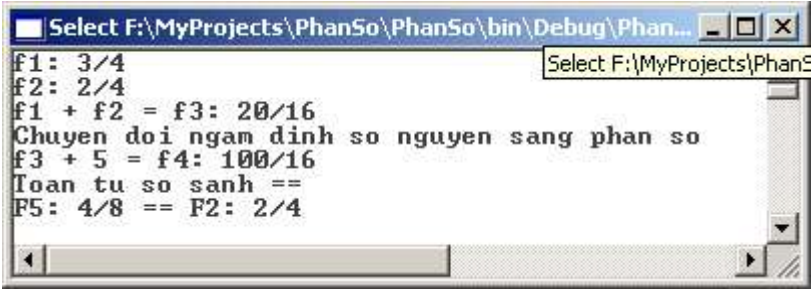
        // ép tự động số 5 thành phân số 5/1 rồi gọi toán tử cộng hai phân số
        PhanSo f4 = f3 + 5;
        Console.WriteLine("f3 + 5 = f4: {0}", f4.ToString());

        PhanSo f5 = new PhanSo(4, 8);
        if (f5 == f2)
            Console.WriteLine("F5: {0} == F2: {1}", f5.ToString(),
f2.ToString());

        Console.ReadLine();
    }
}

```

Kết quả của chương trình:



```

Select F:\MyProjects\PhanSo\PhanSo\bin\Debug\Phan...
f1: 3/4
f2: 2/4
f1 + f2 = f3: 20/16
Chuyen doi ngam dinh so nguyen sang phan so
f3 + 5 = f4: 100/16
Toan tu so sanh ==
F5: 4/8 == F2: 2/4

```

II.12.1. Toán tử với từ khóa checked

Ta có thể dùng từ khóa **checked** để kiểm tra việc tràn số khi định nghĩa một toán tử nào đó. Toán tử này gọi là **checked operator**. Ví dụ nếu cộng 2 số kiểu byte mà cho kết quả lớn hơn 255 thì xảy ra sự tràn số nhưng trình biên dịch của C# vẫn không báo lỗi và làm logic chương trình có thể bị ảnh hưởng.

Ví dụ:

```
public record struct Point(int X, int Y)
{
    // Toán tử có kiểm tra tràn số
    public static Point operator checked +(Point left, Point right)
    {
        checked
        {
            return new Point(left.X + right.X, left.Y + right.Y);
        }
    }

    // Toán tử không kiểm tra tràn số
    public static Point operator +(Point left, Point right)
    {
        return new Point(left.X + right.X, left.Y + right.Y);
    }
}
```

Chú ý:

- ✓ Khi định nghĩa **checked operator** ta phải định nghĩa luôn toán tử tương ứng mà không có từ khóa **check**.
- ✓ Toán tử không có từ khóa **checked** sẽ được thực hiện trong ngữ cảnh khi ta muốn kiểm tra tràn số.
- ✓ Toán tử không có từ khóa **check** sẽ được thực hiện trong ngữ cảnh không cần kiểm tra tràn số.
- ✓ Nếu chỉ có toán tử không có từ khóa **check** thì nó được sử dụng trong cả hai ngữ cảnh.
- ✓ Khi xảy ra sự tràn số **check operator** ném ra ngoại lệ **OverflowException** trong khi toán tử còn lại cho ra kết quả sai (kết quả bị cắt xén).

Bài tập 1: Xây dựng lớp *SoPhuc* (số phức) với các phương thức và các toán tử +, -, *, /, ép sang kiểu số thực...

Bài tập 2: Xây dựng lớp *MaTranVuong* (ma trận vuông) với các phương thức và các toán tử +, -, *, /, ép sang kiểu số thực (trả về định thức của ma trận)...

II.13. Indexer (Chỉ mục)

Việc định nghĩa chỉ mục (indexer) cho phép tạo đối tượng hoạt động giống như một mảng ảo, tức là ta có thể sử dụng toán tử [] để truy cập đến một thành phần dữ liệu nào đó của đối tượng. Chỉ mục không thể là static.

Việc định nghĩa chỉ mục tương tự như việc định nghĩa một thuộc tính.

Cú pháp tạo chỉ mục:

```
public KiểuTraVề this [DanhSáchThamSố]    {
    //Hàm đọc
    get {
        //thân hàm đọc
    }
    // Hàm ghi
    set {
        //thân hàm ghi
    }
}
```

Ví dụ 1: Định nghĩa lớp mảng và dùng indexer để truy cập trực tiếp đến các phần tử của mảng:

```
using System;
class ArrayIndexer {
    int [] myArray;
    int Size;

    public ArrayIndexer (int n)    {
        Size = n;
        myArray = new int[Size];
    }

    // chỉ mục cho phép truy cập đến phần tử thứ index trong mảng.
    public int this [int index]    {
        get {
            // Kiểm tra truy cập ngoài mảng hay không
            if (index < 0 || index >= Size)
                throw new IndexOutOfRangeException();
            else return myArray[index];
        }
        set {
            if (!(index < 0 || index >= Size))
                myArray[index] = value;
        }
    }
}

public class MainClass {
    public static void Main() {
        ArrayIndexer b = new ArrayIndexer(10);
        b[3] = 256;
        b[4] = 1024;
        for (int i=0; i<5; i++)
            Console.WriteLine("b[{0}] = {1}", i, b[i]);
        Console.ReadLine();
    }
}
```

Trong lớp **ArrayIndexer** trên ta định nghĩa một chỉ mục trả về kiểu **int** và có một tham số **index** là chỉ số của phần tử cần truy cập trong mảng:

```
public int this [int index]
```

Phương thức **get** lấy giá trị của phần tử thứ **index**, phương thức **set** gán giá trị cho phần tử thứ **index** trong mảng.

Ví dụ 2:

Giả sử ta cần định nghĩa một lớp có nhiệm vụ thao tác trên file như thao tác trên một mảng các byte. Lớp này có thể hữu dụng khi cần thao tác trên một file rất lớn mà không thể đưa toàn bộ file vào bộ nhớ chính tại một thời điểm, nhất là khi ta chỉ muốn thay đổi một vài byte trong file. Ta định nghĩa lớp **FileByteArray** có chức năng như trên với chỉ mục cho phép truy cập tới byte thứ **index**. Lớp **ReverseApp** sẽ sử dụng một đối tượng thuộc lớp **FileByteArray** để đảo ngược một file.

```
using System;
using System.IO;

// Lớp cho phép xem 1 file như 1 mảng các byte
public class FileByteArray {
    Stream st; //stream dùng để truy cập file

    //Phương thức khởi tạo: mới file và liên kết với stream
    public FileByteArray(string fileName) {
        st = new FileStream(fileName, FileMode .Open);
    }

    // Đóng stream khi thao tác xong
    public void Close() {
        st.Close();
        st = null;
    }

    // Chỉ mục đọc/ghi byte thứ index của file
    public byte this[long index] {
        // Đọc 1 byte tại vị trí index
        get {
            // Tạo vùng đệm chứa dữ liệu
            byte[] buffer = new byte[1];

            // Đưa con trỏ tới vị trí cần đọc
            st.Seek(index, SeekOrigin.Begin);

            // Đọc 1 byte tại vị trí con trỏ, (offset = 0, count = 1)
            st.Read(buffer, 0, 1);
            return buffer[0];
        }

        // Ghi một byte vào file tại vị trí index .
        set {
            //Gán giá trị cần ghi vào buffer
            byte[] buffer = new byte[1] { value };

            //Dịch chuyển đến vị trí cần ghi trong luồng
            st.Seek(index, SeekOrigin.Begin);

            //Ghi dữ liệu vào file

```

```

        st.Write(buffer, 0, 1);
    }
}

// Thuộc tính lay chieu dai cua file (so byte)
public long Length {
    get {
        //Ham seek tra ve vi tri cua con tro.
        return st.Seek(0, SeekOrigin.End);
    }
}
}

public class ReverseApp {
    public static void Main() {
        FileByteArray file;
        file = new FileByteArray("F:\\data.txt");
        long len = file.Length;
        long i;
        // dao nguoc file
        for (i = 0; i < len / 2; ++i) {
            byte t = file[i];
            file[i] = file[len - i - 1];
            file[len - i - 1] = t;
        }
        //Xuat file
        for (i = 0; i < len; ++i) {
            Console.Write((char)file[i]);
        }
        file.Close();
        Console.ReadLine();
    }
}

```

Trong lớp **FileByteArray** ở trên, **chỉ mục** trả về kiểu **byte** và có một tham số **index** là vị trí cần truy cập trong file (kiểu long).

```
public byte this[long index]
```

Phương thức **get** của chỉ mục này định nghĩa các dòng lệnh để đọc một byte từ file, phương thức **set** định nghĩa các dòng lệnh để ghi một byte vào file.

Chú ý:

- Chỉ mục phải có ít nhất một tham số.
- Mặc dù chỉ mục là một đặc điểm thú vị của C# nhưng cần phải sử dụng đúng mục đích (sử dụng để đối tượng có thể hoạt động như mảng hay mảng nhiều chiều).
- Ta có thể định nghĩa chồng nhiều **chỉ mục**.

Bài tập:

Bài tập 1. Xây dựng lớp *Student* với các thuộc tính và chỉ mục dưới đây. Nếu gán giá trị < 0 hoặc > 10 cho điểm toán hoặc điểm tin hoặc gán giá trị cho điểm trung bình sẽ báo lỗi (ném ra biệt lệ) và dừng chương trình.

- ✓ Thuộc tính quản lý điểm toán, điểm tin, điểm trung bình
- ✓ Chỉ mục có tham số là 1 số nguyên để truy cập đến 0 - tên, 1- điểm toán, 2 - điểm tin, 3 - điểm trung bình.
- ✓ Chỉ mục có tham số là 1 chuỗi để truy cập đến "tên" - tên, "điểm toán"- điểm toán, "điểm tin" - điểm tin.

Bài tập 2. Xây dựng lớp đa thức với các toán tử +, -, *, / và chỉ mục để truy cập đến hệ số thứ *i* của đa thức.

II.14. Lớp lồng nhau

Trong C# ta có thể định nghĩa một lớp trong một lớp khác. Điều này cho phép nhóm các lớp có liên quan lại với nhau một cách logic để làm tăng tính đóng gói, làm chương trình dễ đọc, dễ quản lý hơn.

Lớp định nghĩa bên trong một lớp khác gọi là *inner class*, lớp nằm ngoài gọi là *outer class*.

```
using System;
// Outer class
public class Outer_class    {

    public void method1()    {
        Console.WriteLine("Phuong thuc cua Outer class");
    }

    // Inner class
    public class Inner_class    {

        public void method2()    {
            Console.WriteLine("Phuong thuc cua Inner class");
        }
    }
}

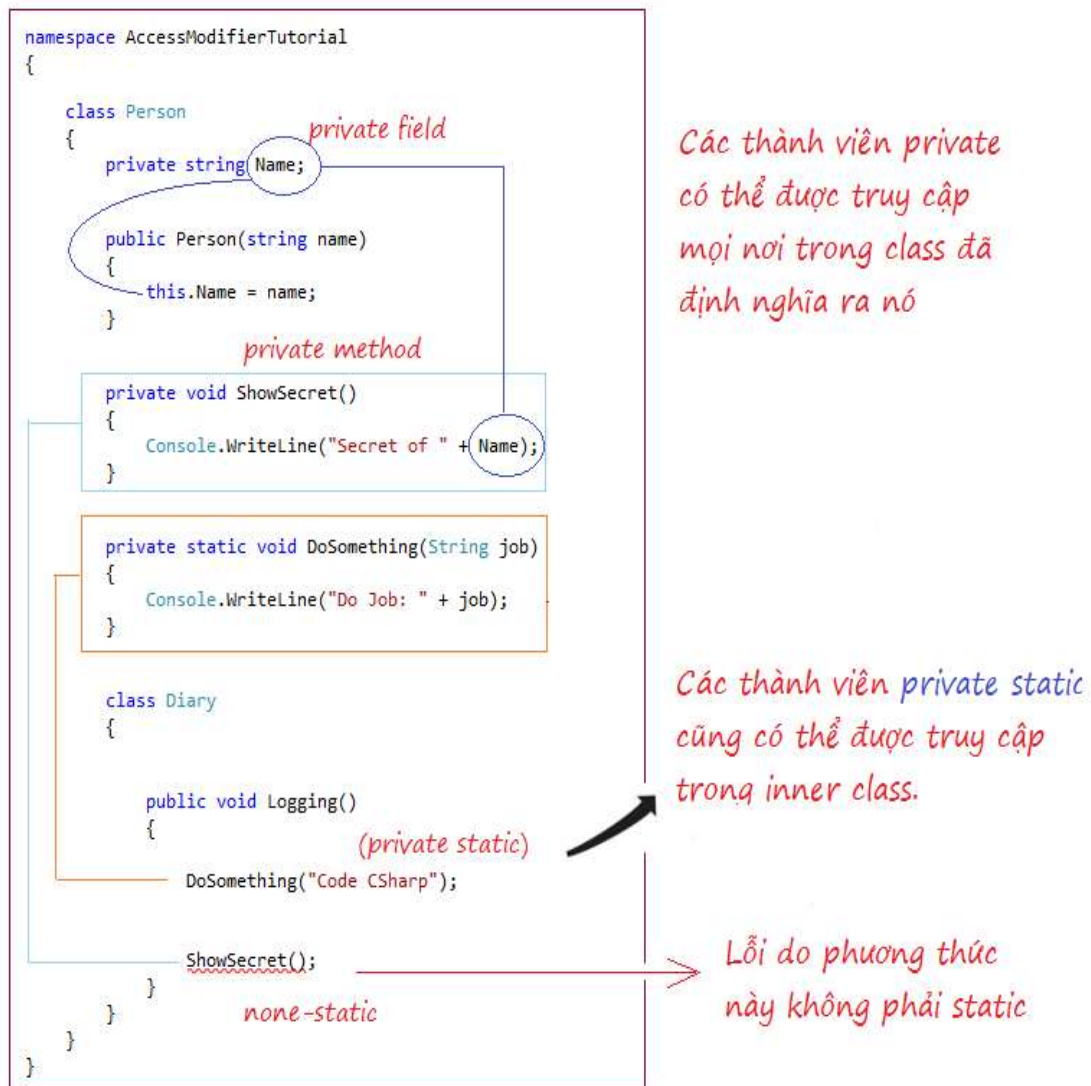
public class App    {

    static public void Main()    {
        // Tạo đối tượng thuộc Outer class
        Outer_class obj1 = new Outer_class();
        obj1.method1();

        // Tạo đối tượng thuộc Outer class
        Outer_class.Inner_class obj2 = new Outer_class.Inner_class();
        obj2.method2();

        Console.Read();
    }
}
```

Các phương thức của *inner class* có thể truy cập trực tiếp đến các thành phần *private static* của *outer class* nhưng phải thông qua một đối tượng như minh họa trong hình sau:



(Nguồn: <https://openplanning.net/10439/csharp-access-modifiers>)

Thông thường *inner class* được tạo ra cho các phương thức của *outer class* sử dụng để thực hiện một số tác vụ nào đó. Lớp lồng được áp dụng để thực hiện tính đa hình. Ta sẽ tìm hiểu kỹ hơn trong phần sau.

II.15. Câu hỏi ôn tập

1. Cú pháp khai báo dữ liệu của lớp?
2. Sự khác nhau giữa thành viên public và các thành viên không phải là public?
3. Có phải tất cả những dữ liệu thành viên luôn luôn được khai báo là public để bên ngoài có thể truy cập chúng?
4. Đối tượng thuộc kiểu dữ liệu tham trị hay tham chiếu?
5. Có thể tạo phương thức bên ngoài của lớp hay không?
6. Sự khác nhau giữa một lớp và một đối tượng của lớp?

7. Thành viên nào trong một lớp có thể được truy cập mà không phải tạo thể hiện của lớp?
8. Khi nào thì phương thức khởi tạo được gọi?
9. Khi nào thì phương thức khởi tạo tĩnh được gọi?
10. Phương thức tĩnh có thể truy cập trực tiếp thành viên nào và không thể truy cập trực tiếp thành viên nào trong một lớp?
11. Có thể truyền biến chưa khởi tạo vào một hàm được không?
12. Làm sao để truyền tham chiếu với biến kiểu giá trị vào trong một phương thức?
13. Sự khác nhau giữa truyền biến tham chiếu và truyền biến tham trị cho hàm?
14. Từ khóa **this** được dùng làm gì trong một lớp?
15. Cú pháp định nghĩa một thuộc tính?
16. Cú pháp định nghĩa một toán tử?
17. Cú pháp định nghĩa một chỉ mục?
18. Nêu ví dụ về cách tạo ra một mảng các đối tượng?
19. Nêu ví dụ về cách truy cập đến một thuộc tính của một đối tượng trong một mảng các đối tượng?
20. Nêu vài nét cơ bản về ArrayList? ArrayList có bị giới hạn về số phần tử có thể lưu trữ không?
21. Có thể lưu các đối tượng thuộc kiểu khác nhau vào chung một ArrayList hay không? Nếu có thì làm sao để biết phần tử thứ i thuộc kiểu nào?
22. Tìm hiểu thêm về Stack, Queue, SortedList

II.16. Bài tập tổng hợp

1. Xây dựng một lớp đường tròn lưu giữ bán kính và tâm của đường tròn. Tạo các thuộc tính cho phép truy cập tới bán kính, chu vi, diện tích của đường tròn.
2. Viết lớp giải phương trình bậc hai. Lớp này có các thuộc tính a , b , c và nghiệm x_1 , x_2 . Lớp cho phép bên ngoài xem được số nghiệm, giá trị các nghiệm của phương trình (nếu có) và cho phép thiết lập hay xem các giá trị a , b , c .
3. Xây dựng lớp ma trận với các phép toán $+$, $-$, $*$, $/$ và chỉ mục để truy cập đến phần tử bất kỳ của ma trận.
4. Xây dựng lớp *Nguoithuebao* (số điện thoại, họ tên), từ đó xây dựng lớp *Danhba* (danh bạ điện thoại) với các phương thức như nhập danh bạ điện thoại, xuất danh bạ điện thoại, tìm số điện thoại theo tên (chỉ mục), tìm tên theo số điện thoại (chỉ mục).

III. Kế thừa (inheritance) và đa hình (polymorphism)

Phần II ở trên đã trình bày cách tạo một kiểu dữ liệu mới bằng các định nghĩa lớp. Việc định nghĩa một lớp thể hiện tính đóng gói và tính trừu tượng của phương pháp lập trình hướng đối tượng. Trong phần này ta sẽ tìm hiểu mối quan hệ giữa các đối tượng và cách thức mô hình hóa những quan hệ này trong mã chương trình dựa trên khái niệm kế thừa và tính đa hình.

III.1. Quan hệ chuyên biệt hóa và tổng quát hóa

Trừ các chương trình nhỏ, các lớp và các đối tượng thường tồn tại trong một mạng các quan hệ và phụ thuộc qua lại lẫn nhau.

Giữa hai loại đối tượng (hai lớp) có thể tồn tại quan hệ tổng quát hóa/chuyên biệt hóa. (Chuyên biệt hóa là mặt đối lập với tổng quát hóa). Quan hệ này tạo nên sự phân cấp hay tạo ra cây quan hệ. Chẳng hạn, quan hệ *is-a* (là một) là một sự chuyên biệt hóa. Ví dụ, khi ta nói “*Sơn dương là một loài động vật, đại bàng cũng là một loài động vật*”, thì có nghĩa là: “*Sơn dương và đại bàng là những loài động vật chuyên biệt, chúng có những đặc điểm chung của động vật và ngoài ra chúng có những đặc điểm phân biệt nhau*”. Và như vậy, **động vật** là tổng quát hóa của **sơn dương** và **đại bàng**; **sơn dương** và **đại bàng** là chuyên biệt hóa của **động vật**.

III.2. Kế thừa

Trong C#, quan hệ chuyên biệt hóa, tổng quát hóa được thể hiện thông qua sự kế thừa. Khi hai lớp có những chức năng, dữ liệu “trùng tự” nhau, ta có thể tách ra các phần chung đó và đưa vào một lớp cơ sở chung cho hai lớp đó để có thể nâng cao khả năng sử dụng lại các mã nguồn chung, cũng như dễ dàng quản lý mã nguồn. Hai lớp đó sẽ kế thừa lớp cơ sở này mà không cần định nghĩa lại các thành phần có trong lớp cơ sở.

Kế thừa là cơ chế cho phép định nghĩa một lớp mới (còn gọi là lớp con, lớp dẫn xuất, derived class) dựa trên một lớp đã có sẵn (còn gọi là lớp cha, lớp cơ sở, base class). Lớp dẫn xuất kế thừa hầu hết các thành phần của lớp cơ sở, bao gồm tất cả các phương thức, biến thành viên của lớp cơ sở và cả những thành phần được kế thừa từ các lớp tổ tiên. Lớp dẫn xuất không kế thừa các phương thức private, **phương thức khởi tạo**, phương thức hủy và phương thức tĩnh của lớp cơ sở.

Cú pháp định nghĩa lớp dẫn xuất:

```
class TênLớpCon : TênLớpCơSở
{
    // Thân lớp dẫn xuất
}
```

Ví dụ: Xây dựng lớp *Point2D* (tọa độ trong không gian 2 chiều), từ đó mở rộng cho lớp *Point3D*.

```
using System;

//Lớp cơ sở Point2D
class Point2D
{
    public int x,y;
    public void Xuat2D()
    {
        Console.WriteLine("{0}, {1}", x, y);
    }
}

//Lớp dẫn xuất Point3D kế thừa từ lớp Point2D
class Point3D: Point2D
{
    public int z;
    public void Xuat3D()
    {
        Console.WriteLine("{0}, {1}, {2}", x, y, z);
    }
}

class PointApp
{
    public static void Main()
    {
        Point2D p2 = new Point2D();
        p2.x = 1;
        p2.y = 2;
        p2.Xuat2D();

        Point3D p3 = new Point3D();
        p3.x = 4;
        p3.y = 5;
        p3.z = 6;
        p3.Xuat3D();
        p3.Xuat2D();
        Console.ReadLine();
    }
}
```

Trong ví dụ trên, rõ ràng trong lớp *Point3D* ta không khai báo các biến *x, y* nhưng phương thức *Xuat3D()* vẫn có thể truy cập *x, y*. Thậm chí trong hàm *Main()*, ta có thể sử dụng đối tượng *p3* để gọi phương thức *Xuat2D()* của lớp cơ sở. Điều này chứng tỏ *Point3D* được kế thừa các biến *x, y* và phương thức *Xuat2D()* từ *Point2D*.

Chú ý:

- Lớp dẫn xuất không thể bỏ đi các thành phần đã được khai báo trong lớp cơ sở.
- Các hàm trong lớp dẫn xuất được truy cập trực tiếp đến hầu hết các thành viên trong lớp cơ sở, trừ các thành viên có mức độ truy cập là *private*. Cần chú ý rằng, lớp dẫn xuất vẫn được kế thừa các thành phần dữ liệu *private* của lớp cơ sở nhưng không được phép truy cập trực tiếp tới chúng (chỉ có thể truy cập gián tiếp thông qua các phương thức của lớp cơ sở).

Trong ví dụ sau đây, câu lệnh $x = x - 1$ sẽ bị báo lỗi “*ClassA.x is inaccessible due to its protection level*” vì x có mức độ truy cập là private nên hàm của ClassB không thể truy cập đến nó.

```
class ClassA
{
    int x = 5;
    public void XuatX()
    {
        Console.WriteLine("{0}", x);
    }
}

class ClassB: ClassA
{
    public void GiamX()
    {
        x = x - 1;    // Lỗi.
    }
}
```

Nếu sửa lại khai báo `int x = 5;` thành `protected int x = 5;` hoặc `public int x = 5;` thì sẽ không còn lỗi trên vì thành phần *protected* hoặc *public* của lớp cơ sở có thể được truy cập trực tiếp trong lớp dẫn xuất (nhưng không được truy cập trong một phương thức không thuộc lớp cơ sở và lớp dẫn xuất).

Hình sau đây tóm tắt sự liên quan giữa kế thừa và các mức độ truy cập:

	Cùng Assembly			Khác Assembly	
	Trong class định nghĩa?	Trong class con	Ngoài class định nghĩa, ngoài class con	Trong class con	Ngoài class con
private	Y				
protected	Y	Y		Y	
internal	Y	Y	Y		
protected internal	Y	Y	Y		
public	Y	Y	Y	Y	Y

(Nguồn: <https://openplanning.net/10439/csharp-access-modifiers>)

III.3. Gọi Phương thức khởi tạo của lớp cơ sở

Vì lớp dẫn xuất không thể kế thừa phương thức khởi tạo của lớp cơ sở nên ta phải định nghĩa phương thức khởi tạo riêng cho lớp dẫn xuất. Nếu lớp cơ sở không có một phương thức khởi tạo hoặc chỉ có phương thức khởi tạo không có tham số thì phương thức khởi tạo của lớp dẫn xuất được định nghĩa như cách thông thường. Nếu lớp cơ sở có phương thức khởi tạo có tham số thì lớp dẫn xuất cũng phải định nghĩa phương thức khởi tạo có tham số theo cú pháp sau:

TênLớpCon(ThamSốLớpCon): TênLớpCơSở(ThamSốLớpCha)

```
{  
    // Khởi tạo giá trị cho các thành phần của lớp dẫn xuất  
}
```

Chú ý: *ThamSốLớpCon* phải bao *ThamSốLớpCha*.

Ví dụ:

```
//Lop co so  
class Point2D {  
    public int x,y;  
    //phuong thuc tao lap cua lop co so co tham so  
    public Point2D(int a, int b) {  
        x = a; y = b;  
    }  
    public void Xuat2D() {  
        Console.WriteLine("{0}, {1}", x, y);  
    }  
}  
  
//Lop dan xuat  
class Point3D:Point2D {  
    public int z;  
  
    // Vi phuong thuc tao lap cua lop co so co tham so  
    // nen phuong thuc tao lap cua lop dan xuat cung phai co tham so  
    public Point3D(int a, int b, int c):base (a,b) {  
        z = c;  
    }  
    public void Xuat3D() {  
        Console.WriteLine("{0}, {1}, {2}", x, y, z);  
    }  
}  
  
class PointApp {  
    public static void Main() {  
        Point2D p2 = new Point2D(1, 2);  
        Console.WriteLine("Toa do cua diem 2 D :");  
        p2.Xuat2D();  
        Console.WriteLine();  
        Point3D p3 = new Point3D(4,5,6);  
        Console.WriteLine("Toa do cua diem 3 D :");  
        p3.Xuat3D();  
        Console.ReadLine();  
    }  
}
```

Trong ví dụ trên, vì phương thức khởi tạo của lớp *Point2D* có tham số:

public Point2D(int a, int b)

nên khi lớp *Point3D* dẫn xuất từ lớp *Point2D*, Phương thức khởi tạo của nó cần có ba tham số. Hai tham số đầu tiên dùng để khởi gán cho các biến *x, y* kế thừa từ lớp *Point2D*, tham số còn lại dùng để khởi gán cho biến thành viên *z* của lớp *Point3D*. Phương thức khởi tạo có nguyên mẫu như sau:

public Point3D(int a, int b, int c):base (a, b)

Phương thức khởi tạo này gọi phương thức khởi tạo của lớp cơ sở **Point2D** bằng cách đặt dấu “:” sau danh sách tham số và từ khoá **base** với các đối số tương ứng với phương thức khởi tạo của lớp cơ sở.

III.4. Định nghĩa phiên bản mới trong lớp dẫn xuất

Qua những phần trên chúng ta có nhận xét rằng, khi cần định nghĩa hai lớp mà chúng có chung một vài đặc trưng, chức năng thì những thành phần đó nên được đặt vào một lớp cơ sở. Sau đó hai lớp này sẽ kế thừa từ lớp cơ sở đó và bổ sung thêm các thành phần của riêng chúng. Ngoài ra, lớp dẫn xuất còn có quyền định nghĩa lại các phương thức đã kế thừa từ lớp cơ sở nhưng không còn phù hợp với nó nữa.

Lớp dẫn xuất kế thừa hầu hết các thành viên của lớp cơ sở vì vậy trong bất kỳ phương thức nào của lớp dẫn xuất ta có thể truy cập trực tiếp đến các thành viên này (mà không cần thông qua một đối tượng thuộc lớp cơ sở). Tuy nhiên, nếu lớp dẫn xuất cũng có một thành phần **X** (biến hoặc phương thức) nào đó trùng tên với thành viên thuộc lớp cơ sở thì trình biên dịch sẽ có cảnh báo (warning) dạng như sau:

```
"keyword new is required on 'LớpDẫnXuất.X' because  
it hides inherited member on 'LớpCơSở.X' "
```

Bởi vì, khi trong lớp dẫn xuất khai báo một thành phần trùng tên với một thành phần trong lớp cơ sở thì trình biên dịch hiểu rằng người dùng muốn che dấu các thành viên của lớp cơ sở nên nó yêu cầu người dùng đặt từ khóa **new** ngay câu lệnh khai báo thành phần đó để đảm bảo đây không phải là một nhầm lẫn vô tình. Điều này có tác dụng che dấu thành phần kế thừa đó đối với các phương thức bên ngoài lớp dẫn xuất. Nếu phương thức của lớp dẫn xuất muốn truy cập đến thành phần **X** của lớp cơ sở thì phải sử dụng từ khóa **base** theo cú pháp:

base.X.

Ví dụ:

```
using System;  
  
class XeHoi  
{  
    //Cac thanh phan nay la protected de phuong thuc Xuat cua lop XeXat va  
    XeHoi co the truy cap duoc  
  
    protected int TocoDo;  
    protected string BienSo;  
    protected string HangSX;  
  
    public XeHoi(int td, string BS, string HSX)  
    {  
        TocoDo = td; BienSo = BS; HangSX = HSX;  
    }  
  
    public void Xuat()  
    {  
        Console.WriteLine("Xe: {0}, Bien so: {1}, Toc do: {2} kmh", HangSX, BienSo,  
TocoDo);
```

```
    }  
}  
  
class XeCar : XeHoi  
{  
    int SoHanhKhach;  
  
    public XeCar(int td, string BS, string HSX, int SHK)  
        : base(td, BS, HSX)  
    {  
        SoHanhKhach = SHK;  
    }  
  
    //Tu khoa new che dau phuong thuc Xuat cua lop XeHoi vi phuong thuc Xuat  
    //cua lop XeHoi khong con phu hop voi lop XeCar.  
    public new void Xuat()  
    {  
        // Goi phuong thuc xuat cua lop co so thong qua tu khoa base  
        base.Xuat();  
        Console.WriteLine(", {0} cho ngoi", SoHanhKhach);  
    }  
}  
  
class XeTai : XeHoi  
{  
    int TrongTai;  
  
    public XeTai(int td, string BS, string HSX, int TT)  
        : base(td, BS, HSX)  
    {  
        TrongTai = TT;  
    }  
  
    // Tu khoa new che dau phuong thuc Xuat cua lop XeHoi  
    // vi phuong thuc Xuat cua lop XeHoi khong con phu hop voi lop XeCar nua.  
    public new void Xuat()  
    {  
        base.Xuat(); // Goi phuong thuc xuat cua lop co  
  
        Console.WriteLine(", trong tai {0} tan", TrongTai);  
    }  
}  
  
public class Test  
{  
    public static void Main()  
    {  
        XeCar c = new XeCar(150, "49A-4444", "Toyota", 24);  
        c.Xuat();  
        XeTai t = new XeTai(150, "49A-5555", "Benz", 12);  
        t.Xuat();  
        Console.ReadLine();  
    }  
}
```

Trong ví dụ trên, lớp **XeHoi** có một phương thức *Xuat()*, tuy nhiên phương thức này chỉ xuất ra các thông tin như **BienSo**, **TocDo**, **HangSX** nên không còn phù

hợp với hai lớp *XeTai* và *XeCar* nữa. Do đó hai lớp dẫn xuất này định nghĩa một phiên bản mới của phương thức *Xuat()* theo cú pháp sau:

```
public new void Xuat()
{
    ...
}
```

Việc thêm vào từ khoá **new** chỉ ra rằng người lập trình muốn tạo ra một phiên bản mới của phương thức này trong các lớp dẫn xuất nhằm che dấu phương thức đã kế thừa từ lớp cơ sở *XeHoi*. Như vậy, trong hàm **Main()**, khi gọi:

```
c.Xuat();
```

hoặc

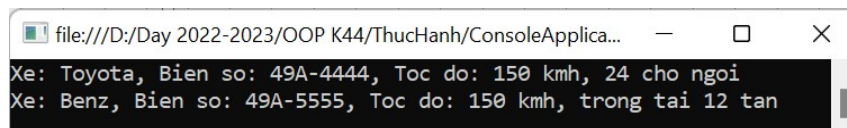
```
t.Xuat();
```

trình biên dịch sẽ hiểu rằng đây là phương thức *Xuat()* của lớp *XeTai* hoặc lớp *XeCar*.

Hơn nữa, trong phương thức *Xuat()* của lớp *XeTai* và *XeCar* ta vẫn có thể gọi phương thức *Xuat()* của lớp *XeHoi* bằng câu lệnh:

```
base.Xuat();
```

Kết quả chạy chương trình:




```
file:///D:/Day 2022-2023/OOP K44/ThucHanh/ConsoleApplica...
Xe: Toyota, Bien so: 49A-4444, Toc do: 150 kmh, 24 cho ngoi
Xe: Benz, Bien so: 49A-5555, Toc do: 150 kmh, trong tai 12 tan
```

III.5. Tham chiếu thuộc lớp cơ sở

Một tham chiếu thuộc lớp cơ sở có thể trỏ đến một đối tượng thuộc lớp dẫn xuất nhưng nó chỉ được phép truy cập đến các thành phần được khai báo trong lớp cơ sở. Với các lớp *XeHoi*, *XeCar* như trên, ta có thể định nghĩa hàm **Main()** như sau:

```
public static void Main()
{
    XeCar c = new XeCar(150, "49A-4444", "Toyota", 24);
    c.Xuat();
    Console.WriteLine();
    Console.WriteLine("Tham chieu cua lop co so XeHoi co the tro den doi tuong thuoclop dan xuat XeCar");
    Console.WriteLine("Nhưng chỉ có thể gọi ham xuat tuong ung voi XeHoi");
    XeHoi h = c;
    h.Xuat();
    Console.ReadLine();
}
```

Kết quả chạy chương trình:



```
F:\MyProjects\Vehicle\Vehicle\bin\Debug\Vehicle.exe
Xe: Toyota, Bien so: 49A-4444, Toc do: 150 kmh, 24 cho ngoi
Tham chieu cua lop co so XeHoi co the tro den doi tuong thuoclop dan xuat XeCar
Nhưng chỉ có thể gọi hàm xuất tương ứng với XeHoi
Xe: Toyota, Bien so: 49A-4444, Toc do: 150 kmh
```

Khi gọi lệnh

`c.Xuat() ;`

trình biên dịch gọi phương thức *Xuat()* của lớp *XeCar* và xuất các thông tin: *hàng sản xuất (Toyota), biển số (49A-444), tốc độ tối đa (150 km/h), 24 chỗ ngồi*.

Sau đó một đối tượng *h* thuộc lớp cơ sở *XeHoi* trở đến đối tượng *c* thuộc lớp dẫn xuất :

`XeHoi h = c;`

Khi gọi lệnh

`h.Xuat() ;`

trình biên dịch sẽ thực hiện phương thức *Xuat()* của lớp *XeHoi* nên chỉ xuất các thông tin: *hàng sản xuất (Toyota), biển số (49A-444), tốc độ tối đa (150 km/h)*.

Bài tập 1: Xây dựng lớp *Stack* và lớp *Queue* (cài đặt theo kiểu danh sách liên kết) bằng cách đưa những thành phần dữ liệu và phương chung của hai lớp này vào một lớp cơ sở *SQ* và từ đó xây dựng các lớp *Stack*, *Queue* kế thừa từ lớp *SQ*.

Bài tập 2: Xây dựng lớp hình tròn với các thuộc tính (properties): bán kính, đường kính, diện tích.

Xây dựng lớp hình cầu kế thừa từ lớp hình tròn. Lớp này che dấu đi các thuộc tính: diện tích (dùng từ khóa *new*) đồng thời bổ sung thêm thuộc tính: thể tích.

- Diện tích hình cầu tính bán kính R được tính theo công thức $4 \cdot \pi \cdot R^2$
- Thể tích hình cầu tính bán kính R được tính theo công thức $\frac{4}{3} \cdot \pi \cdot R^3$

Bài tập 3: Tương tự, xây dựng lớp hình trụ tròn kế thừa từ lớp hình tròn với các thuộc tính: chu vi mặt đáy, diện tích mặt đáy, diện tích xung quanh, diện tích toàn phần, thể tích.

III.6. Phương thức ảo (virtual method) và tính đa hình (polymorphism)

Hai đặc điểm mạnh nhất của kế thừa đó là khả năng sử dụng lại mã chương trình và đa hình (polymorphism). Đa hình là ý tưởng “sử dụng một giao diện chung cho nhiều phương thức khác nhau”, dựa trên phương thức ảo (virtual method) và cơ chế liên kết muộn (late binding). Nói cách khác, đây là cơ chế cho phép gọi một loại thông điệp tới nhiều đối tượng khác nhau mà mỗi đối tượng lại có cách xử lý riêng theo ngữ cảnh tương ứng của chúng.

Ví dụ, ta có thể cài đặt chung một thuật toán sắp xếp nổi bọt cho các loại dữ liệu số nguyên, số thực, ký tự, chuỗi, hay cho bất kỳ các loại đối tượng có thể so sánh thứ tự với nhau. Trong trường hợp này, việc so sánh thứ tự của hai đối tượng mang tính đa hình vì mỗi loại đối tượng sẽ có cách so sánh phù hợp. Ở đây ưu điểm của tính đa hình thể hiện ở chỗ ta có thể định nghĩa trước thuật toán sắp xếp nổi bọt mà không quan tâm quá chi tiết đến loại dữ liệu nào sẽ được sử dụng để sắp xếp, miễn là hai phần tử bất kỳ có thể so sánh thứ tự với nhau.

Sau đây là một kịch bản khác thực hiện tính đa hình:

Lớp của các đối tượng nút nhấn (*button*), nhãn (*label*), nút chọn (*option button*), danh sách sổ xuống (*combobox*)... đều kế thừa từ lớp *Window* và đều có các phương thức vẽ (hiển thị) riêng của mình lên form. Một form có thể có nhiều đối tượng như trên và được lưu trong một danh sách (không cần biết các đối tượng trong danh sách là *ListBox* hay *Button*... miễn là đối tượng đó là một thể hiện *Window*). Khi form được mở, nó có thể yêu cầu mỗi đối tượng *Window* tự vẽ lên form bằng cách gọi thông điệp vẽ đến từng đối tượng trong danh sách và các đối tượng này sẽ thực hiện chức năng vẽ tương ứng. Khi đó ta muốn *form* xử lý tất cả các đối tượng *Window* theo đặc trưng đa hình.

Để thực hiện được đa hình ta phải thực hiện các bước sau:

1. Lớp cơ sở đánh dấu phương thức ảo bằng từ khóa **virtual** hoặc **abstract**.

2. Các lớp dẫn xuất định nghĩa lại phương thức ảo này (đánh dấu bằng từ khóa *override*).
3. Vì tham chiếu thuộc lớp cơ sở có thể trỏ đến một đối tượng thuộc lớp dẫn xuất và có thể truy cập hàm ảo đã định nghĩa lại trong lớp dẫn xuất nên khi thực thi chương trình, tùy đối tượng được tham chiếu này trỏ tới mà phương thức tương ứng được gọi thực hiện. Nếu tham chiếu này trỏ tới đối tượng thuộc lớp cơ sở thì phương thức ảo của lớp cơ sở được thực hiện. Nếu tham chiếu này trỏ tới đối tượng thuộc lớp dẫn xuất thì phương thức ảo đã được lớp dẫn xuất đã định nghĩa lại được thực hiện.

Ví dụ:

```
using System;
public class MyWindow
{
    protected int top, left;    //Toa do goc tren ben trai
    public MyWindow(int t, int l)
    {
        top = t;
        left = l;
    }

    // Phuong thuc ao
    public virtual void DrawWindow()
    {
        Console.WriteLine("...dang ve Window tai toa do {0}, {1}", top, left);
    }
}

public class MyListBox : MyWindow
{
    string listBoxContents;
    public MyListBox(int top, int left, string contents)
        : base(top, left)
    {
        listBoxContents = contents;
    }

    public override void DrawWindow()
    {
        Console.WriteLine("...dang ve listbox {0} tai toa do: {1},{2}",
listBoxContents, top, left);
    }
}

public class MyButton : MyWindow
{
    public MyButton(int top, int left) : base(top, left) { }
    public override void DrawWindow()
    {
        Console.WriteLine("...dang ve button tai toa do: {0},{1}", top, left);
    }
}

public class Tester
{
    static void Main()
```



```

{
    Random R = new Random();
    int t;
    string s = "";
    MyWindow[] winArray = new MyWindow[4];
    for (int i = 0; i < 4; i++)
    {
        t = R.Next() % 3;
        switch (t)
        {
            case 0:
                winArray[i] = new MyWindow(i * 2, i * 4); break;
            case 1:
                s = "thu " + (i + 1).ToString();
                winArray[i] = new MyListBox(i * 3, i * 5, s);
                break;
            case 2:
                winArray[i] = new MyButton(i * 10, i * 20); break;
        }
    }

    for (int i = 0; i < 4; i++)
    {
        winArray[i].DrawWindow();
    }
    Console.ReadLine();
}
}

```

Trong ví dụ này ta xây dựng một lớp *MyWindow* có một phương thức ảo:

```
public virtual void DrawWindow( )
```

Các lớp *MyListBox*, *MyButton* kế thừa từ lớp *MyWindow* và định nghĩa lại (**override**) phương thức *DrawWindow()* theo cú pháp:

```
public override void DrawWindow( )
```

Sau đó trong hàm *Main()* ta khai báo và tạo một mảng các đối tượng *MyWindow*. Vì mỗi phần tử thuộc mảng này là một tham chiếu thuộc lớp *MyWindow* nên nó có thể trỏ tới bất kỳ một đối tượng nào thuộc các lớp kế thừa lớp *MyWindow*, chẳng hạn lớp *MyListBox* hay lớp *MyButton*.

Vòng lặp *for* đầu tiên tạo ngẫu nhiên các đối tượng thuộc một trong các lớp *MyWindow*, *MyListBox*, *MyButton*, vì vậy, tại thời điểm biên dịch chương trình, trình biên dịch không biết đối tượng thứ *i* thuộc lớp nào và do đó chưa thể xác định được đoạn mã của phương thức *DrawWindow()* cần gọi. Tuy nhiên, tại thời điểm chạy chương trình, sau vòng lặp *for* đầu tiên, mỗi *winArray[i]* tham chiếu tới một loại đối tượng cụ thể nên trình thực thi sẽ tự động xác định được phương thức *DrawWindow()* cần gọi. (Như vậy ta đã sử dụng một giao diện chung là *DrawWindow()* cho nhiều phương thức khác nhau).

Chú ý rằng nếu phương thức *DrawWindow()* trong các lớp *MyListBox*, *MyButton*,... không có từ khóa **override** như cú pháp:

```
public override void DrawWindow( )
```

thì trình biên dịch sẽ báo lỗi.

Ví dụ 2:

Một điểm dịch vụ cần quản lý các thông tin cho thuê xe đạp và xe máy. Với xe đạp cần lưu họ tên người thuê, số giờ thuê. Tiền thuê xe đạp được tính như sau: 10000 (đồng) cho giờ đầu tiên, 80000 cho mỗi giờ tiếp theo. Với mỗi xe máy cần lưu họ tên người thuê, số giờ thuê, loại xe (100 phân khối, 250 phân khối), biển số. Tiền thuê xe máy được tính như sau: Đối với giờ đầu tiên, loại xe 100 phân khối tính 15000; loại xe 250 phân khối tính 20000. Đối với những giờ tiếp theo tính 10000 cho cả hai loại xe máy.

Viết chương trình xây dựng các lớp cần thiết sau đó nhập danh sách các thông tin thuê xe đạp và xe máy, sau đó xuất ra các thông tin sau:

- Xuất tất cả các thông tin thuê xe (cả số tiền thuê tương ứng).
- Tính tổng số tiền cho thuê xe đạp và xe máy.
- Xuất tất cả các thông tin liên quan đến việc thuê xe đạp.
- Tính tổng số tiền cho thuê xe máy loại 250 phân khối.

Mã chương trình:

```
using System;
public class XE
{
    protected string hoten;
    protected int giothue;
    public virtual int TienThue
    {
        get { return 0; }
    }
    public virtual void Xuat() { }
    public virtual string ID()
    {
        return "X";
    }
}

public class XEDAP : XE
{
    public XEDAP(string ht, int gt)
    {
        hoten = ht;
        giothue = gt;
    }

    public override string ID()
    {
        return "XD";
    }

    public override int TienThue
    {
        get
        {
            int T = 10000;
            if (giothue > 0) T = T + 80000 * (giothue - 1);
            return T;
        }
    }
}
```

```

    }
}
public override void Xuat()
{
    Console.WriteLine(hoten + "\t" + giothue + "\t" + TienThue);
}
}
public class XEMAY : XE
{
    string loaixe;
    string bienso;
    public XEMAY(string ht, int gt, string lx, string bs)
    {
        hoten = ht;
        giothue = gt;
        loaixe = lx;
        bienso = bs;
    }
    public override string ID()
    {
        if (loaixe == "100") return "XM_100";
        else return "XM_250";
    }

    public override int TienThue
    {
        get
        {
            int T;
            if (loaixe == "100") T = 15000;
            else
            {
                T = 20000;
                if (giothue > 0) T = T + 10000 * (giothue - 1);
            }
            return T;
        }
    }
    public override void Xuat()
    {
        Console.WriteLine("Xe may: " + hoten + "\t" + giothue + "\t" + loaixe
+ "\t" + bienso + "\t" + TienThue);
    }
}
public class CUAHANG
{
    public int n;
    XE[] XT; //Xe cho thue

    public CUAHANG(int size)
    {
        n = size;
        XT = new XE[size];
    }
    public int menu()
    {
        int chon;
        do
        {

```

```

        Console.WriteLine("***** Bang Chon Nhap *****");
        Console.WriteLine("1. Nhap Xe Dap");
        Console.WriteLine("2. Nhap Xe May");
        chon = int.Parse(Console.ReadLine());
    } while ((chon != 1) && (chon != 2));
    return chon;
}
public void NhapDSXeChoThue()
{
    int chon;
    for (int i = 0; i < n; i++)
    {
        chon = menu();
        if (chon == 1)
        {
            Console.WriteLine("***** Ban chon nhap xe dap *****");
            Console.WriteLine("Ho ten nguoi thue?");
            string ht = Console.ReadLine();
            Console.WriteLine("So gio thue?");
            int gt = int.Parse(Console.ReadLine());
            XT[i] = new XEDAP(ht, gt);
        }
        else
        {
            Console.WriteLine("***** Ban chon nhap xe may *****");
            Console.WriteLine("Nguoi thue?");
            string ht = Console.ReadLine();
            Console.WriteLine("So gio thue?");
            int gt = int.Parse(Console.ReadLine());
            Console.WriteLine("Ban nhap vao loai xe(100 hoac 125 )phan
khoi:");
            string lx = Console.ReadLine();
            Console.WriteLine("Bien so:");
            string bs = Console.ReadLine();
            XT[i] = new XEMAY(ht, gt, lx, bs);
        }
    }
}
public void XuatDSXeThue()
{
    for (int i = 0; i < n; i++)
    {
        XT[i].Xuat();
    }
}
public long TongTienChoThue()
{
    long ts = 0;
    for (int i = 0; i < n; i++)
    {
        ts = ts + XT[i].TienThue;
    }
    return ts;
}
public void XuatXeDap()
{
    for (int i = 0; i < n; i++)
    {
        if (XT[i].ID() == "XD") XT[i].Xuat();
    }
}

```

```
    }  
}  
  
public long TongTienXeMay250()  
{  
    long T = 0;  
    for (int i = 0; i < n; i++)  
    {  
        if (XT[i].ID() == "XM_250") T += XT[i].TienThue;  
    }  
    return T;  
}  
}  
public class App  
{  
    public static void Main()  
    {  
        CUAHANG ch = new CUAHANG(3);  
        ch.NhapDSXeChoThue();  
        Console.WriteLine("Xuat tat ca nhung thong tin thue xe");  
        ch.XuatDSXeThue();  
  
        Console.WriteLine("Tong tien thue xe " + ch.TongTienChoThue());  
        Console.WriteLine("Thong tin ve xe dap cho thue:");  
        ch.XuatXeDap();  
  
        Console.WriteLine("Tong tien thue xe may 250 phan khoi" +  
ch.TongTienXeMay250());  
        Console.ReadLine();  
    }  
}
```

Kết quả chạy chương trình:

```

file:///D:/Day 2022-2023/OOP K44/ThucHanh/ConsoleApplication8/bin/
***** Bang Chon Nhap *****
1. Nhap Xe Dap
2. Nhap Xe May
1
***** Ban chon nhap xe dap *****
Ho ten nguoi thue?
A
So gio thue?
2
***** Bang Chon Nhap *****
1. Nhap Xe Dap
2. Nhap Xe May
2
***** Ban chon nhap xe may *****
Nguoi thue?
B
So gio thue?
3
Ban nhap vao loai xe(100 hoac 125 )phan khoi:
100
Bien so:
bx50
***** Bang Chon Nhap *****
1. Nhap Xe Dap
2. Nhap Xe May
1
***** Ban chon nhap xe dap *****
Ho ten nguoi thue?
C
So gio thue?
1
Xuat tat ca nhung thong tin thue xe
A      2      18000
Xe may: B      3      100      bx50      15000
C      1      10000
Tong tien thue xe 43000
Thong tin ve xe dap cho thue:
A      2      18000
C      1      10000
Tong tien thue xe may 250 phan khoi0

```

Bài tập.

Bài tập 1. Sửa lại lớp CuaHang trong Ví dụ 2/trang 50 (cho thuê xe đạp, xe máy): Lưu trữ các xe cho thuê bằng ArrayList thay vì dùng mảng các đối tượng.

Bài tập 2. Một công ty bảo hiểm cần quản lý thông tin về các nhân viên cũng như các bảo hiểm mà các nhân viên bán được (bảo hiểm dài hạn và bảo hiểm ngắn hạn). Với nhân viên, cần quản lý: tên, hệ số lương, các bảo hiểm bán được.

Với loại bảo hiểm ngắn hạn cần quản lý: tên người mua, thời hạn (mấy tháng), to số tiền phải đóng. Nhân viên sẽ được hưởng tiền hoa hồng bằng 5% số tiền. Với loại bảo hiểm dài hạn cần quản lý: tên người mua, thời hạn (trên 12 tháng), số tiền phải đóng, số tiền đóng hàng tháng. Nhân viên sẽ được hưởng tiền hoa hồng bằng 50% số tiền đóng 1 tháng.

Ngoài ra công ty quy định:

- ✓ Thưởng 100 USD cho những nhân viên có bán ít nhất một bảo hiểm trên 10000 USD.
- ✓ Nhân viên nào có tổng số tiền bán được < 10000 USD sẽ bị phạt 30 USD.
- ✓ Lương của nhân viên được tính theo công thức:
- ✓ $40 * \text{hệ số lương} + 1\% \text{ tổng số tiền của các bảo hiểm nhân viên đó bán được.}$

Hãy xây dựng các lớp đối tượng tương ứng để quản lý cho công ty trên và viết một ứng dụng có nhiệm vụ như sau:

- Nhập danh sách các nhân viên và các bảo hiểm mà họ bán được.
- Xuất thông tin của các nhân viên (gồm cả lương) và các bảo hiểm mà họ bán được.
- Xuất danh sách những nhân viên có tiền hoa hồng > 50USD.
- Xuất danh sách những nhân viên bị phạt tiền.
- Xuất danh sách những nhân viên được thưởng 100USD.
- Sắp xếp danh sách nhân viên theo tổng số tiền bán được.

Bài tập 3. ???

III.7. Lớp Object

Tất cả các lớp trong C# đều được dẫn xuất từ lớp **Object**. Lớp **Object** là lớp gốc trên cây phân cấp kế thừa, nó cung cấp một số phương thức mà các lớp con có thể ghi đè (**override**) như:

Phương thức	Ý nghĩa
<i>Equals()</i>	Kiểm tra hai đối tượng có tương đương nhau không
<i>GetHashCode()</i>	Cho phép đối tượng cung cấp hàm Hash riêng để sử dụng trong kiểu tập hợp.
<i>GetType()</i>	Trả về kiểu đối tượng.
<i>ToString()</i>	Trả về chuỗi biểu diễn đối tượng.
<i>Finalize()</i>	Xóa đối tượng trong bộ nhớ.
<i>MemberwiseClone()</i>	Tạo copy của đối tượng.

Trong những ví dụ trước ta đã thực hiện việc ghi đè lên phương thức *ToString()* và *Equals()* của lớp **Object**.

III.8. Lớp trừu tượng(**abstract**)

Trong các ví dụ về phương thức ảo trên, nếu lớp dẫn xuất không định nghĩa lại phương thức ảo của lớp cơ sở thì nó được kế thừa như thông thường. Tức là lớp dẫn xuất không nhất thiết phải định nghĩa lại phương thức ảo.

Để bắt buộc tất cả các lớp dẫn xuất phải định nghĩa lại (**override**) một phương thức của lớp cơ sở ta phải đặt từ khóa **abstract** trước phương thức đó và phương thức đó được gọi là phương thức trừu tượng. Trong phần thân của phương thức trừu tượng không có câu lệnh nào, nó chỉ tạo một tên phương thức và đánh dấu rằng nó phải được thi hành trong lớp dẫn xuất.

Lớp chứa phương thức trừu tượng được gọi là lớp trừu tượng. Phương thức trừu tượng phải được đặt trong lớp trừu tượng. Lớp trừu tượng có từ khóa **abstract** đứng trước.

Có hai điểm quan trọng cần chú ý về lớp trừu tượng:

- ✓ Các lớp kế thừa một lớp trừu tượng nhưng không định nghĩa lại các phương thức trừu tượng đã kế thừa thì chúng cũng là các lớp trừu tượng.
- ✓ Ta chỉ có thể khai báo các biến thuộc lớp trừu tượng nhưng không thể tạo ra một đối tượng thực sự thuộc lớp trừu tượng.

Cú pháp khai báo phương thức trừu tượng:

abstract public void TênPhươngThức(...tham số);

Ví dụ: Xây dựng lớp **HìnhHoc** với phương thức tính chu vi, diện tích là phương thức trừu tượng hoặc phương thức ảo. Sau đó định nghĩa các lớp **HìnhChuNhat** (hình chữ nhật), **HìnhTron** (hình tròn) kế thừa từ lớp **HìnhHoc** với các thành phần dữ liệu và phương thức tính chu vi, diện tích cụ thể của từng loại đối tượng.

```
using System;
// lop hình học (truu tuong)
abstract public class HìnhHoc
{
    abstract public double DienTich();
    virtual public double ChuVi() { return 0; }
}

// lop hình tron kế thừa từ lop hình học
public class HìnhTron : HìnhHoc
{
    double _bankinh;
    public double BanKinh
    {
        get { return _bankinh; }
        set { _bankinh = value; }
    }

    public override double DienTich()
    {
        return _bankinh * _bankinh * 3.1416;
    }

    public override double ChuVi()
    {
        return _bankinh * 2 * 3.1416;
    }
}

// lop hình chu nhật kế thừa từ lop hình học
public class HìnhChuNhat : HìnhHoc
{
    double _dai, _rong;

    public double ChieuDai
    {
        get { return _dai; }
        set { _dai = value; }
    }

    public double ChieuRong
    {
        get { return _rong; }
    }
}
```



```
        set { _rong = value; }
    }

    public override double DienTich() { return _dai * _rong; }
    public override double ChuVi() { return (_dai + _rong) * 2; }
}

class Tester
{
    static void Main(string[] args)
    {
        HìnhHoc h;
        // h = new HìnhHoc(); //---> Lỗi
        HìnhTron t = new HìnhTron();
        t.BanKinh = 5;
        Console.WriteLine("Thông tin về hình tron");
        h = t;
        Console.WriteLine("Chu vi hình tron: {0} ", h.ChuVi());
        Console.WriteLine("Diện tích hình tron:{0} ", h.DienTich());

        HìnhChuNhat n = new HìnhChuNhat();
        n.ChieuDai = 4;
        n.ChieuRong = 3;
        h = n;
        Console.WriteLine("Thông tin về hình chu nhật ");
        Console.WriteLine("Chu vi hình chu nhật:{0}", h.ChuVi());
        Console.WriteLine("Diện tích hình chu nhật:{0}", h.DienTich());
        Console.ReadLine();
    }
}
```

Trong lớp **HìnhHoc** ở ví dụ trên, ta khai báo phương thức tính diện tích là một phương thức trừu tượng (không có phần cài đặt mã của phương thức vì chưa biết đối tượng hình thuộc dạng nào nên không thể tính diện tích của nó). Như vậy, lớp **HìnhHoc** là một lớp trừu tượng.

abstract public double DienTich();

Trong lớp **HìnhHoc** trên, ta cũng khai báo một phương thức tính chu vi là phương thức ảo với mục đích minh họa rằng trong lớp trừu tượng có thể có phương thức ảo (hoặc bất cứ một thành phần nào khác của một lớp). Vì đây là một phương thức không trừu tượng nên phải có phần cài đặt mã bên trong thân phương thức.

Các lớp **HìnhChuNhat**, **HìnhTron** kế thừa từ lớp **HìnhHoc** nên chúng buộc phải cài đặt lại phương thức tính diện tích.

Trong hàm **Main()**, ta tạo ra đối tượng **n** thuộc lớp **HìnhChuNhat** và đối tượng **t** thuộc lớp **HìnhTron**. Khi tham chiếu **h** thuộc lớp **HìnhHoc** trỏ tới đối tượng **n** (tương tự với đối tượng **t**), nó có thể gọi được phương thức tính diện tích của lớp **HìnhChuNhat** (tương tự lớp **HìnhTron**), điều này chứng tỏ phương thức trừu tượng cũng có thể dùng với mục đích đa hình.

Kết quả chạy chương trình:

```
file:///D:/Day 2022-2023/OOP K44
Thông tin về hình tròn
Chu vi hình tròn: 31.416
Diện tích hình tròn: 78.54
Thông tin về hình chữ nhật
Chu vi hình chữ nhật: 14
Diện tích hình chữ nhật: 12
```

Chú ý: Phân biệt giữa từ khóa *new* và *override*

- Từ khóa ***override*** dùng để định nghĩa lại (ghi đè) phương thức ảo (***virtual***) hoặc phương thức trừu tượng (***abstract***) của lớp cơ sở, nó được dùng với mục đích đa hình.
- Từ khóa ***new*** để che dấu thành viên của lớp cơ sở trùng tên với thành viên của lớp dẫn xuất.

III.9. Giao diện (*interface*)

Giao diện là một dạng của lớp trừu tượng được sử dụng với mục đích hỗ trợ tính đa hình. Trong giao diện không có bất cứ một cài đặt nào, chỉ có nguyên mẫu của các phương thức, chỉ mục, thuộc tính mà một lớp khác khi kế thừa nó thì phải có cài đặt cụ thể cho các thành phần này. Tức là lớp kế thừa giao diện tuyên bố rằng nó hỗ trợ các phương thức, thuộc tính, chỉ mục được khai báo trong giao diện. Khi một lớp kế thừa một giao diện ta nói rằng lớp đó thực thi (implement) giao diện.

Khác với lớp trừu tượng, giao diện bắt buộc các lớp kế thừa nó phải định nghĩa lại các thành phần trừu tượng của giao diện, nếu không sẽ xảy ra lỗi biên dịch. Tức là, khi kế thừa một giao diện ta phải thực thi mọi phương thức, thuộc tính, ... của giao diện.

III.9.1. Thực thi giao diện

Ta dùng từ khóa ***interface*** để khai báo một giao diện với cú pháp sau:

```
[MứcĐộ TruyCập]           interface           TênGiaoDiện
[:CácGiaoDiệnCơSở]
{
    Nội dung
}
```

MứcĐộ TruyCập: *public* hoặc *internal*.

CácGiaoDiệnCơSở: danh sách các *interface* khác mà nó kế thừa.

Về mặt cú pháp, một giao diện giống như một lớp chỉ có (các) phương thức trừu tượng. Giao diện có thể chứa khai báo của phương thức, thuộc tính, chỉ mục, sự kiện (events) nhưng không được chứa biến dữ liệu.

Chú ý:

- Mặc định, tất cả các thành phần khai báo trong giao diện đều là *public* mà không cần phải khai báo thêm từ khóa *public*. (Nếu có từ khóa *public* đứng trước sẽ bị báo lỗi). Các thành phần trong giao diện chỉ là các khai báo, không có phần cài đặt mã.
- Một lớp có thể kế thừa một lớp khác đồng thời kế thừa nhiều giao diện.

Ví dụ: Mọi chiếc xe hơi hoặc xe máy đều có hành động (phương thức) khởi động và dừng. Ta có thể dùng định nghĩa một giao diện kèm thêm một thuộc tính để cho biết chiếc xe đã khởi động hay chưa:

```
public interface IDrivable
{
    void KhởiDong() ;
    void Dung() ;

    bool DaKhởiDong
    {
        get;
    }
}
```

Thuộc tính đã khởi động (*DaKhởiDong*) chỉ có phương thức *get* vì khi gọi phương thức *KhởiDong()* thì thuộc tính này sẽ có giá trị *true*, khi gọi phương thức *Dung()* thì thuộc tính này sẽ có giá trị *false*.

Khi đó, một lớp *Car* thực thi giao diện này phải cài đặt các phương thức và thuộc tính đã khai báo trong giao diện *IDrivable* trên.

Mã đầy đủ của chương trình minh họa như sau:

```
public interface IDrivable
{
    void KhởiDong() ;
    void Dung() ;
    bool DaKhởiDong
    {
        get;
    }
}

public class Car: IDrivable
{
    private bool Started = false;
    public void KhởiDong()
    {
        Console.WriteLine("Xe ca khoi dong");
        Started = true;
    }
}
```

```
    }  
    public void Dung()  
    {  
        Console.WriteLine("Xe ca dung");  
        Started = false;  
    }  
    public bool DaKhoiDong  
    {  
        get{return Started;}  
    }  
}  
public class Tester  
{  
    public static void Main()  
    {  
        Car c = new Car();  
        c.KhoiDong();  
        Console.WriteLine("c.DaKhoiDong      =      "      +  
c.DaKhoiDong);  
        c.Dung();  
        Console.WriteLine("c.DaKhoiDong      =      "      +  
c.DaKhoiDong);  
        Console.ReadLine();  
    }  
}
```

III.9.2. Hủy đối tượng

Ta đã biết rằng khi tạo ra một đối tượng bằng toán tử new thì đối tượng được cấp phát trên bộ nhớ HEAP và phương thức khởi tạo được gọi thực hiện để khởi gán các giá trị cho đối tượng. Khi đối tượng hết hiệu lực nó cần giải phóng vùng nhớ đã được cấp phát, một phương thức gọi là phương thức hủy (hàm hủy *Finalizer* hay *Destructor*) của đối tượng nó sẽ tự động thi hành.

.NET có cơ chế thu dọn rác tự động, thông qua đối tượng GC (garbage collection), có chức năng định kỳ kiểm tra và thu hồi vùng nhớ của các đối tượng cấp phát trên HEAP khi chúng không còn được sử dụng nữa. Nếu ta không định nghĩa phương thức hủy thì phương thức hủy mặc định sẽ được dùng và việc thu hồi vùng nhớ của các đối tượng do GC đảm trách. Do vậy, thông thường ta không cần phải quan tâm đến việc giải phóng vùng nhớ.

Tuy nhiên, nếu lớp (đối tượng) có các thành phần không được quản lý như các đối tượng con dùng để thao tác trên file, kết nối mạng hay kết nối cơ sở dữ liệu thì GC không kiểm soát được. Khi đó, lớp cần có phương thức hủy để giải phóng các tài nguyên của thành phần đó khi đối tượng thuộc lớp hết hiệu lực.

C# cung cấp phương thức *Finalize()* để thực hiện công việc này. Phương thức *Finalize()* của đối tượng sẽ được gọi ngầm thông qua phương thức hủy, bởi GC, khi đối tượng hết hiệu lực. GC duy trì một danh sách những đối tượng có phương

thức **Finalize()**. Các đối tượng hết hiệu lực sẽ được đặt vào hàng đợi của GC và định kỳ GC sẽ giải phóng vùng nhớ của các đối tượng này.

Có ba cách thu hồi vùng nhớ của các đối tượng đã hết hiệu lực sau:

a) Định nghĩa phương thức hủy

Ta khai báo một phương thức hủy trong C# như sau:

```
~Class1()
{
    // Thực hiện một số công việc giải phóng bộ nhớ
}
```

Đoạn lệnh trên được ngầm hiểu như sau:

```
Class1.Finalize()
{
    // Thực hiện một số công việc giải phóng bộ nhớ
    base.Finalize();
}
```

Ví dụ sau minh họa phương thức *Finalize()* tự động được gọi.

```
using System;

class NhanVien
{
    private string ten;

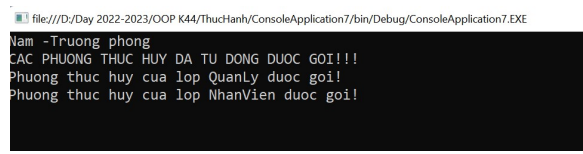
    public string Ten
    {
        get { return ten; }
        set { ten = value; }
    }

    public NhanVien(string ten)
    {
        this.ten = ten;
    }

    // Destructor: phương thức hủy
    ~NhanVien()
    {
        //Ta gán các biến tham chiếu = null để giải phóng vùng nhớ
        ten = null;
        Console.WriteLine("Phương thức hủy của lớp NhanVien được gọi! ");
    }
}
```

```
}  
}  
  
class QuanLy : NhanVien  
{  
    // Constructor  
    private string chucVu;  
  
    public string ChucVu  
    {  
        get { return chucVu; }  
        set { chucVu = value; }  
    }  
    public QuanLy(string ten, string chucVu): base(ten)  
    {  
        this.chucVu = chucVu;  
    }  
  
    // Destructor  
    ~QuanLy()  
    {  
        // Ta gan cac bien tham chieu = null để hủy vùng nhớ  
        chucVu = null;  
        Console.WriteLine("Phuong thuc huy cua lop QuanLy duoc goi!");  
    }  
}  
  
class Program2  
{  
    static void Main(string[] args)  
    {  
        Test();  
        Console.WriteLine("CAC PHUONG THUC HUY DA TU ĐỘNG ĐƯỢC GOI!!!");  
        Console.Read();  
    }  
  
    private static void Test()  
    {  
        // Chỉ khởi tạo đối tượng của lớp QuanLy  
        QuanLy t = new QuanLy("Nam", "Truong phong");  
        Console.WriteLine(t.Ten + " -" + t.ChucVu);  
  
        // Sau lệnh này đối tượng của lớp QuanLy mất hiệu lực vì không còn biến nào  
        // tham chiếu đến nó nữa  
        t = null;  
        System.GC.Collect();  
    }  
}
```

Kết quả chạy chương trình:



```
file:///D:/Day 2022-2023/OOP K44/ThucHanh/ConsoleApplication7/bin/Debug/ConsoleApplication7.EXE  
Nam -Truong phong  
CAC PHUONG THUC HUY DA TU ĐỘNG ĐƯỢC GOI!!!  
Phuong thuc huy cua lop QuanLy duoc goi!  
Phuong thuc huy cua lop NhanVien duoc goi!
```

Trong hàm *Test()* ta khởi tạo một đối tượng của lớp *QuanLy* và cho biến *t* trỏ tới. Ở cuối hàm, ta cố ý gán *t = null* để đối tượng này mất hiệu lực vì không còn biến nào tham chiếu đến nó nữa. Sau đó ta gọi lệnh để yêu cầu .Net giải phóng vùng nhớ đã cấp phát

cho đối tượng này. Tuy nhiên, kết quả chạy chương trình như hình trên cho thấy lệnh `System.GC.Collect()` chỉ ghi nhận sẽ giải phóng vùng nhớ của đối tượng nhưng không thực hiện ngay. Khi kết thúc hàm `Test()`, chương trình còn thực hiện thêm một lệnh bên ngoài hàm này (lệnh `Console.WriteLine("CAC PHUONG THUC HUY DA TU DONG DUOC GOI!!!")`) rồi mới gọi thực hiện các phương thức `~QuanLy()` của lớp `QuanLy`.

Do cơ chế kế thừa, trước khi kết thúc phương thức `~QuanLy()` thì phương thức `~NhanVien()` cũng được thực hiện để giải phóng các vùng nhớ của các thành phần kế thừa từ lớp `NhanVien`.

b) Dùng phương thức `Dispose`

Như chúng ta đã biết thì việc gọi một phương thức kết thúc `Finalize()` trong C# là không hợp lệ, vì phương thức này được thực hiện một cách tự động bởi bộ thu dọn rác.

Nếu muốn đóng hay giải phóng vùng nhớ của các tài nguyên không kiểm soát được của một đối tượng bất cứ lúc nào ta muốn thì ta nên thực thi giao diện `IDisposable` và định nghĩa lại (override) phương thức `Dispose()` của nó.

Giao diện `IDisposable` được hỗ trợ bởi cơ chế dọn rác của .NET (ngay tức thì) mà không phải chờ cho đến khi phương thức `Finalize()` được gọi.

Khi định nghĩa phương thức `Dispose()` thì phải ngưng bộ thu dọn bằng phương thức `Finalize()`. Để thực hiện điều đó gọi lệnh `GC.SuppressFinalize(this)`, trong đó, tham chiếu `this` chính là đối tượng đang cần giải phóng vùng nhớ.

Phương thức `Dispose()` có cấu trúc như sau:

```
public void Dispose()
{
    // Thực hiện công việc dọn dẹp
    // Yêu cầu bộ thu dọn GC trong thực hiện kết thúc
    GC.SuppressFinalize( this );
}
```

Ví dụ:

```
using System;

public class Mang : IDisposable
{
    int[] Data;
    public int Size;

    public Mang(int n)
    {
        Data = new int[n];
        Size = n;
    }
}
```

```

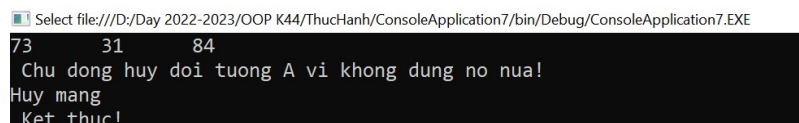
public void Dispose()
{
    // Thuc hien viec don dep
    Console.WriteLine("Huy mang");
    Data = null;
    Size = 0;
    // yeu cau bo thu don GC thuc hien ket thuc
    GC.SuppressFinalize(this);
}

public void TaoNgauNhien()
{
    Random r = new System.Random();
    for (int i = 0; i < Size; i++)
    {
        Data[i] = r.Next(1,10);
    }
}

public void Xuat()
{
    for (int i = 0; i < Size; i++)
    {
        Console.Write("{0} \t", Data[i]);
    }
}
}
class App
{
    static void Main(string[] args)
    {
        Mang A = new Mang(5);
        A.TaoNgauNhien();
        A.Xuat();
        Console.WriteLine("\n Chu dong huy doi tuong A vi khong dung no nua!");
        A.Dispose();
        Console.WriteLine(" Ket thuc!");
        Console.ReadLine();
    }
}

```

Kết quả chạy chương trình như hình dưới đây cho thấy việc giải phóng vùng nhớ khi đối tượng A bị hủy được thực hiện ngay khi gọi lệnh A.Dispose() mà không cần chờ đến khi kết thúc chương trình.



The screenshot shows a Windows command prompt window with the title "Select file:///D:/Day 2022-2023/OOP K44/ThucHanh/ConsoleApplication7/bin/Debug/ConsoleApplication7.EXE". The output of the program is as follows:

```

73      31      84
Chu dong huy doi tuong A vi khong dung no nua!
Huy mang
Ket thuc!

```

c) Dùng chỉ thị using

Để việc quên gọi phương thức **Dispose()** khi cần thiết ta có thể sử dụng kèm với chỉ thị **using**. Chỉ thị này đảm bảo phương thức **Dispose()** tự động được gọi khi kết thúc khối lệnh được bao bởi cặp dấu **{}** của chỉ thị **using**.

Ví dụ 1:

```

class App
{

```



```

static void Main(string[] args)
{
    Mang A = new Mang(3);
    A.TaoNgauNhiem();
    A.Xuat();
    Console.WriteLine("\n Chu dong huy doi tuong A vi khong dung no nua!");
    A.Dispose();

    using (Mang B = new Mang(5))
    {
        B.TaoNgauNhiem();
        B.Xuat();
        Console.WriteLine("\n Doi tuong B bi huy vi het hieu luc.");
    }

    Console.WriteLine(" Ket thuc!");
    Console.ReadLine();
}
}

```

Kết quả chạy chương trình như hình dưới đây cho thấy đối tượng A bị hủy (giải phóng vùng nhớ) ngay tức thì khi thực hiện lệnh `A.Dispose()` và khi đối tượng B tự động bị hủy khi kết thúc phạm vi của chỉ thị `using`.

```

file:///D:/Day 2022-2023/OOP K44/ThucHanh/ConsoleApplication7/bin/Debug/ConsoleApplication7.EXE
52      94      51
Chu dong huy doi tuong A vi khong dung no nua!
Huy mang
56      53      91      36      47
Doi tuong B bi huy vi het hieu luc!
Huy mang
Ket thuc!

```

Tham khảo thêm về việc hủy đối tượng tại: <https://learn.microsoft.com/en-us/dotnet/api/system.object.finalize?view=net-7.0>

III.9.3. Thực thi nhiều giao diện

Các lớp có thể thực thi nhiều giao diện, đây là cách để thực hiện đa kế thừa trong C#.

Ví dụ: Tạo một giao diện tên là **IStorable** với các phương thức **Write()** để lưu nội dung của đối tượng vào file và phương thức **Read()** để đọc dữ liệu từ file. Sau đó ta tạo lớp **Document** thực thi giao diện **IStorable** để các đối tượng thuộc lớp này có thể đọc từ cơ sở dữ liệu hoặc lưu trữ vào cơ sở dữ liệu. Việc mở file được thực hiện thông qua đối tượng **fs** thuộc lớp **FileStream**, việc ghi và đọc file thông qua đối tượng thuộc các lớp **StreamWriter** và **StreamReader**. Đồng thời lớp **Document** cũng thực thi một giao diện khác tên là **Iencryptable**, giao diện này có hai phương thức là mã hóa (**Encrypt()**) và giải mã (**Decrypt()**):

```

using System;
using System.IO;

// Khai bao giao dien
interface IStorable
{
    // mac dinh cac khai bao phuong thuc la public. Khong cai dat gi
    void Read(string FileName);
}

```

```
void Write(string FileName);
string Data { get; set; }
}

interface IEncryptable
{
    void Encrypt();
    void Decrypt();
}

// Lop Document thuc thi giao dien IStorable
public class Document : IStorable, IEncryptable
{
    string S;
    public Document(string str)
    {
        S = str;
    }
    // thuc thi phuong thuc Read cua giao dien IStorable
    public void Read(string FileName)
    {
        //Mo fiel de doc
        using (FileStream fs = new FileStream(FileName, FileMode.Open))
        {
            //tao luong doc du lieu
            StreamReader sr = new StreamReader(fs);
            //Doc tung dong cho den khi gia tri doc duoc la null
            string text;
            S = "";
            while ((text = sr.ReadLine()) != null)
            {
                S = S + text;
            }
            //Dong luong va dong file
            sr.Close();
            fs.Close();
        }
    }
    // thuc thi phuong thuc Write cua giao dien IStorable
    public void Write(string FileName)
    {
        //Mo file de ghi du lieu
        using (FileStream fs = new FileStream(FileName, FileMode.OpenOrCreate))
        {
            //Tao luong ghi du lieu vao file
            StreamWriter sw = new StreamWriter(fs);
            //ghi chuoí S ra file
            sw.WriteLine(S);
            //dong luong va dong file
            sw.Close();
            fs.Close();
        }
    }
    // thuc thi thuoc tinh Data cua giao dien IStorable
    public string Data
    {
        get { return S; }
        set { S = value; }
    }
}
```

```

// thực thi phương thức Encrypt của giao diện IEncryptable
public void Encrypt()
{
    string KQ = "";
    for (int i = 0; i < S.Length; i++)
        KQ = KQ + (char)((int)S[i] + 5);
    S = KQ;
}
// thực thi phương thức Decrypt của giao diện IEncryptable
public void Decrypt()
{
    string KQ = "";
    for (int i = 0; i < S.Length; i++)
        KQ = KQ + (char)((int)S[i] - 5);
    S = KQ;
}
}
// Thu nghiệm chương trình
public class Tester
{
    static void Main()
    {
        string FileName = "D:\\datafile.txt";
        Document doc = new Document("THIEU NU la viet tat cua tu THIEU NU
TINH");
        doc.Write(FileName);
        doc.Read(FileName);
        Console.WriteLine("Du lieu goc: {0}", doc.Data);
        Console.WriteLine("Du lieu da ma hoa:");
        doc.Encrypt();
        Console.WriteLine(doc.Data);
        Console.WriteLine("Du lieu sau khi giai ma:");
        doc.Decrypt();
        Console.WriteLine(doc.Data);
        Console.ReadLine();
    }
}

```

Vì giao diện là một dạng lớp cơ sở (trừu tượng) nên biến tham chiếu kiểu giao diện có thể trỏ tới một đối tượng thuộc lớp thực thi giao diện và gọi những phương thức cần thiết của giao diện thông qua biến này. Chẳng hạn, trong hàm *Main()* trên ta có thể thay câu lệnh:

```
doc.Read(FileName);
```

bằng hai câu lệnh sau:

```

IStorable isDoc = doc; //(IStorable) doc;
isDoc.Read(FileName);

```

Kết quả chạy chương trình:

```

file:///D:/Day 2022-2023/OOP K44/ThucHanh/ConsoleApplication8/bin/
Du lieu goc: THIEU NU la viet tat cua tu THIEU NU TINH
Du lieu da ma hoa:
YMNJZ%SZ%qf%{njy%yfy%hzf%yz%YMNJZ%SZ%YNSM
Du lieu sau khi giai ma:
THIEU NU la viet tat cua tu THIEU NU TINH

```

III.9.4. Mở rộng giao diện

Ta cũng có thể mở rộng giao diện bằng cách bổ sung những thành viên hay phương thức mới. Chẳng hạn, ta có thể mở rộng giao tiếp *IStorable* thành *IStorableAndCompressible* bằng cách kế thừa từ giao tiếp *IStorable* bổ sung các phương thức nén file và giải nén file:

```

interface IStorableAndCompressible : IStorable
{
    // bo sung them phuong thuc nen va giai nen
    void Compress ( );
    void Decompress ( );
}

```

III.9.5. Kết hợp giao diện

Thay vì lớp *Document* thực thi hai giao diện *IStorable*, *IEncryptable*, ta có thể tạo kết hợp hai giao diện này thành một giao diện mới là *IStorableAndEncryptable*. Sau đó lớp *Document* thực thi giao diện mới này:

```

interface IStorableAndEncryptable: IStorable,
IEncryptable
{
    //Có thể bổ sung thêm các phương thức, thuộc tính... mới
}
public class Document : IStorableAndEncryptable
{
    ...
}

```

III.9.6. Kiểm tra đối tượng có hỗ trợ giao diện hay không bằng toán tử *is*

Toán tử *is* dùng để kiểm tra một đối tượng có thuộc về một lớp/giao diện nào đó hay không. Chú ý rằng, một đối tượng có thuộc về một lớp con thì cũng thuộc về một lớp cha. Do đó ta có thể kiểm tra một đối tượng có hỗ trợ (thi công) giao diện hay không bằng cách dùng toán tử *is* (để gọi các thực hiện các phương thức đã cài đặt lại của giao diện đó hay thực hiện ép kiểu...). Nếu đối tượng không hỗ trợ giao diện mà ta cố tình ép kiểu thì sẽ xảy ra biệt lệ.

Ví dụ: kiểm tra đối tượng *doc* có hỗ trợ giao diện *IStorable* hay không, nếu có thì ép sang kiểu *IStorable* và gọi phương thức *Read()*.

```

static void Main ( )
{

```

```

Document doc = new Document("Test Document");
// chỉ ep sang kiểu IStorable neu doi tuong co ho tro
if (doc is IStorable)
{
    IStorable isDoc = (IStorable) doc;
    isDoc.Read( );
}
}

```

III.9.7. Các giao diện IComparer, IComparable và ArrayList

Lớp ArrayList và các giao diện IComparer, IComparable thuộc name space System.Collections.

a) ArrayList

ArrayList có các chức năng của cả mảng và danh sách liên kết. Bảng sau XXX đây tóm tắt một số phương thức và thuộc tính của ArrayList.

Tên	Loại	Ý nghĩa
Count	Thuộc tính	Số phần tử hiện có
Add	Hàm	Thêm một phần tử vào cuối
BinarySearch	Hàm	Tìm kiếm nhị phân
Contains	Hàm	Kiểm tra một phần tử có thuộc Array List hay không
Clear	Hàm	Xóa các phần tử
IndexOf	Hàm	Trả về vị trí đầu tiên của phần tử cần tìm
LastIndexOf	Hàm	Trả về vị trí cuối cùng của phần tử cần tìm
Insert	Hàm	Chèn 1 phần tử vào 1 vị trí nào đó
Remove	Hàm	Loại bỏ 1 phần tử
RemoveAt	Hàm	Loại bỏ 1 phần tử tại một vị trí nào đó
Reverse	Hàm	Đảo chiều
Sort	Hàm	Sắp xếp
...	...	

Ví dụ 1:

- Khai báo và cấp phát đối tượng ArrayList AL:

```
ArrayList AL = new ArrayList();
```

- Thêm một phần tử có giá trị 5 vào cuối AL:

```
AL.Add(5);
```

- Chèn giá trị 6 vào đầu AL:

```
AL.Insert(0, 6);
```

- Xuất phần tử thứ 2 trong AL:

```
Console.WriteLine("{0}", AL[1]);
```

b) Giao diện IComparer và giao diện IComparable

Hai giao diện này hỗ trợ việc gọi các phương thức cần tới việc so sánh hai đối tượng của các lớp thư viện của C#.

- Lớp kế thừa giao diện IComparer cần phải cài đặt phương thức so sánh hai đối tượng obj1 và obj2 của một lớp nào đó:

```
public int Compare(Object obj1, Object obj2)
```

- Lớp kế thừa giao diện IComparable cần phải cài đặt phương thức so sánh của đối tượng ngầm định thuộc lớp do con trỏ *this* trỏ tới với một đối tượng obj2 khác (thường thuộc cùng lớp).

```
public int CompareTo(Object obj2)
```

c) Sắp xếp, tìm kiếm đối tượng dựa trên các hàm có sẵn của ArrayList.

Việc sắp xếp hay tìm kiếm đối tượng đã được cài đặt sẵn thành các phương thức **Sort**, **BinarySearch** của **ArrayList** như trong bảng ???. Các hàm này hoạt động dựa trên sự so sánh phần tử trong ArrayList. (Hai phần tử có bằng nhau không? Phần tử nào nhỏ hơn?). Nếu các phần tử này thuộc kiểu nguyên thủy như int, float, char,... thì đã có cách so sánh mặc định. Tuy nhiên, nếu các phần tử trong ArrayList là các đối tượng thì ta phải chỉ ra cách so sánh hai đối tượng (nếu không sẽ gây ngoại lệ: “**System.InvalidOperationException was unhandled. Failed to compare two elements in the array**” khi chạy chương trình).

Về cơ bản, có hai cách để so sánh hai đối tượng (để các hàm **Sort**, **BinarySearch** của **ArrayList** có thể hoạt động được):

- **Cách 1.** Lớp của đối tượng phải cài đặt giao diện **IComparable** và phải cài đặt lại phương thức **public int CompareTo(Object rhs)** để so sánh đối tượng ngầm định **this** với đối tượng **rhs** của lớp. Xem Ví dụ 2 và Ví dụ 3.
- **Cách 2.** Việc so sánh hai đối tượng trong **ArrayList** phải nhờ vào một đối tượng (gọi là **thăm tử** hoặc **trọng tài**) thuộc về một lớp trung gian. Lớp trung gian đó phải kế thừa giao diện **IComparer** và định nghĩa lại phương thức **public int Compare(Object lhs, Object rhs)** để có thể so sánh đối tượng trong **ArrayList**. Khi gọi các phương thức **Sort**, **BinarySearch** ta phải truyền tham số cho chúng một đối tượng thuộc lớp trung gian này. Xem Ví dụ 4.

Nguyên mẫu của các phương thức **Sort, **BinarySearch** của **ArrayList** như sau:**

<code>BinarySearch(Object key)</code>	<p>Tìm vị trí phần tử key trong toàn bộ ArrayList theo phương pháp tìm kiếm nhị phân. Sử dụng cách so sánh mặc định của các phần tử trong ArrayList. Nói cách khác, nếu các phần tử trong ArrayList là các đối tượng thì lớp của đối tượng phải cài đặt giao diện IComparable như Cách 1.</p> <p>Các phần tử trong ArrayList cần được sắp xếp tăng dần. Nếu không thấy sẽ trả về -1.</p>
<code>BinarySearch(Object key, IComparer detective)</code>	<p>Tìm key trong toàn bộ ArrayList theo phương pháp tìm kiếm nhị phân như phương thức trên nhưng việc so sánh xem phần tử nào trong ArrayList bằng với key dựa vào cách so sánh của một đối tượng detective (thăm tử). Lớp của detective kế thừa giao diện IComparer như Cách 2.</p>
<code>BinarySearch(Int32, Int32, Object, IComparer)</code>	<p>Tìm key trong toàn bộ ArrayList theo phương pháp tìm kiếm nhị phân như phương thức trên nhưng việc so sánh dựa vào cách so sánh trong một phạm vi.</p>
<code>Sort()</code>	<p>Sắp xếp các phần tử trong toàn bộ ArrayList theo phương pháp QuickSort, sử dụng cách so sánh mặc định của các phần tử trong ArrayList. Nói cách khác, nếu các phần tử trong ArrayList là các đối tượng thì lớp của đối tượng phải cài đặt giao diện IComparable như Cách 1.</p>
<code>Sort(IComparer referee)</code>	<p>Sắp xếp các phần tử trong toàn bộ ArrayList theo phương pháp QuickSort. Việc so sánh thứ tự giữa hai phần tử trong ArrayList dựa vào cách so sánh của một đối tượng referee (trọng tài). Lớp của referee kế thừa giao diện IComparer như Cách 2.</p>
<code>Sort(Int32, Int32, IComparer)</code>	<p>Như Cách 2 nhưng chỉ sắp xếp trong một phạm vi.</p>

Ví dụ 2: sắp xếp trên ArrayList theo tên của nhân viên.

```
using System;
using System.Collections;

// Lớp NhanVien thực thi giao diện IComparable
class NhanVien : IComparable
{
    string Ten;
    long Luong;
```

```
public NhanVien(string T, long L)
{
    Ten = T;
    Luong = L;
}

public override string ToString()
{
    return Ten + ", " + Luong;
}

// Định nghĩa lại hàm public int CompareTo(Object rhs) của IComparable
// Hàm này so sánh 2 đối tượng NhanVien theo Ten
public int CompareTo(Object rhs)
{
    NhanVien r = (NhanVien)rhs; // Ép kiểu Obliect sang NhanVien
    // return Ten.CompareTo(r.Ten);
}
}

class App
{
    static void Main(string[] args)
    {
        ArrayList empArray = TaoCacNhanVien();

        Console.WriteLine("=====");
        Console.WriteLine("Cac nhan vien");
        XuatCacNhanVien(empArray);

        //Chen 2 nhan vien vao vi tri thu 2 và 4
        NhanVien nv = new NhanVien("Anh Tu", 900);
        empArray.Insert(1, nv);
        nv = new NhanVien("Chi Tu", 300);
        empArray.Insert(3, nv);

        //Xuat
        Console.WriteLine("=====");
        Console.WriteLine("Sau khi them 2 nhan vien");
        XuatCacNhanVien(empArray);
        Console.WriteLine("Tat ca co {0} nhan vien", empArray.Count);

        // Xóa nhân viên cuối cùng
        Console.WriteLine("=====");
        empArray.RemoveAt(3);
        // Sắp xếp theo tên
        empArray.Sort();
        Console.WriteLine("Sau khi xoa nhan vien thu 4 va sap xep theo Ten:");
        XuatCacNhanVien(empArray);
        Console.WriteLine("Tat ca co {0} nhan vien", empArray.Count);

        Console.ReadLine();
    }

    private static void XuatCacNhanVien(ArrayList empArray)
    {
        foreach (NhanVien x in empArray) Console.WriteLine(x.ToString());

        Console.WriteLine("Tat ca co {0} nhan vien", empArray.Count);
    }
}
```



```

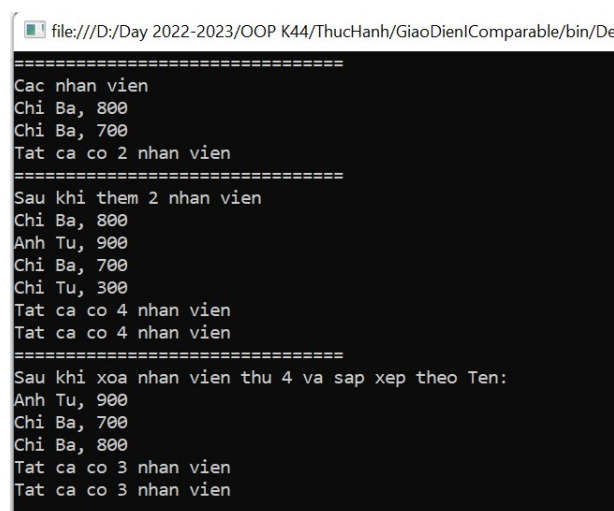
    }

    private static ArrayList TaoCacNhanVien()
    {
        string[] AT = { "Anh Hai", "Chi Hai", "Anh Ba", "Chi Ba" };
        long[] AL = { 500, 600, 700, 800 };

        ArrayList empArray = new ArrayList();
        Random r = new Random(10);
        string T;
        long L;
        //Tạo 2 nhân viên có và Lương lấy ngẫu nhiên từ 2 mảng trên
        for (int i = 0; i < 2; i++)
        {
            T = AT[r.Next(4)];
            L = AL[r.Next(4)];
            empArray.Add(new NhanVien(T, L));
        }
        return empArray;
    }
}

```

Kết quả chạy chương trình (như Hình XXX) cho thấy các nhân viên đã được sắp xếp theo tên, dù rằng ta không trực tiếp cài đặt thuật toán sắp xếp.



```

file:///D:/Day 2022-2023/OOP K44/ThucHanh/GiaoDienIComparable/bin/De
=====
Cac nhan vien
Chi Ba, 800
Chi Ba, 700
Tat ca co 2 nhan vien
=====
Sau khi them 2 nhan vien
Chi Ba, 800
Anh Tu, 900
Chi Ba, 700
Chi Tu, 300
Tat ca co 4 nhan vien
Tat ca co 4 nhan vien
=====
Sau khi xoa nhan vien thu 4 va sap xep theo Ten:
Anh Tu, 900
Chi Ba, 700
Chi Ba, 800
Tat ca co 3 nhan vien
Tat ca co 3 nhan vien

```

d) Linh động trong cách quy định tiêu chí sắp xếp

Trong thực tế có thể có nhiều tiêu chí để so sánh các đối tượng với nhau. Tùy thời điểm hay yêu cầu mà ta cần sắp xếp hoặc tìm kiếm theo tiêu chí tương ứng. Ví dụ, lúc thì ta muốn sắp xếp (so sánh) thứ tự các nhân viên theo tên, lúc ta lại muốn sắp xếp theo lương, hoặc cả hai. Khi đó ta cần có một biến để lưu giữ tiêu chí sắp xếp hiện tại và định nghĩa phương thức **Compare** (nếu kế thừa giao diện **IComparer**) hay phương thức **CompareTo** (nếu kế thừa giao diện **IComparable**) tương ứng với biến này. Đồng thời, trước khi sắp xếp ta cần đặt giá trị biến để lưu giữ tiêu chí sắp xếp một cách tương ứng như trong ví dụ sau đây.

Ví dụ 3.

```

using System;
using System.Collections;

```

```
// Lớp NhanVien thực thi giao diện IComparable
class NhanVien : IComparable
{
    string Ten;
    long Luong;

    public static String TieuchiSapXep="Ten";

    public NhanVien(string T, long L)
    {
        Ten = T;
        Luong = L;
    }

    public override string ToString()
    {
        return Ten + ", " + Luong;
    }

    // Định nghĩa lại hàm public int CompareTo(Object rhs) của IComparable
    // Hàm này so sánh 2 đối tượng NhanVien theo Ten
    public int CompareTo(Object rhs)
    {
        NhanVien r = (NhanVien)rhs; // Ép kiểu Object sang NhanVien
        if (TieuchiSapXep=="Ten") return Ten.CompareTo(r.Ten);
        else // Biểu thức này sai do lỗi tràn số nếu giá trị của Luong nhân
viên quá lớn
            return (int) (Luong - r.Luong);
    }
}

class App
{
    static void Main(string[] args)
    {
        ArrayList empArray = TaoCacNhanVien(5);
        Console.WriteLine("=====");
        Console.WriteLine("Danh sach cac nhan vien:");
        XuatCacNhanVien(empArray);

        // Sắp xếp theo tên (mặc định)
        empArray.Sort();
        Console.WriteLine("\nSau khi sap xep theo Ten:");
        XuatCacNhanVien(empArray);

        // Sắp xếp theo lương
        NhanVien.TieuchiSapXep = "Luong";
        empArray.Sort();
        Console.WriteLine("\nSau khi sap xep theo Luong:");
        XuatCacNhanVien(empArray);

        Console.ReadLine();
    }

    private static void XuatCacNhanVien(ArrayList empArray)
    {
        foreach (NhanVien x in empArray) Console.WriteLine(x.ToString());
    }
}
```

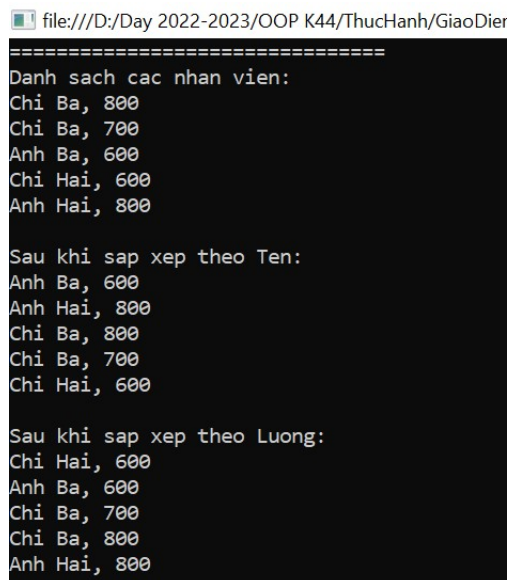
```

private static ArrayList TaoCacNhanVien(int SoNV)
{
    string[] AT = { "Anh Hai", "Chi Hai", "Anh Ba", "Chi Ba" };
    long[] AL = { 500, 600, 700, 800 };

    ArrayList empArray = new ArrayList();
    Random r = new Random(10);
    string T;
    long L;
    //Tạo 2 nhân viên có và Lương lấy ngẫu nhiên từ 2 mảng trên
    for (int i = 0; i < SoNV; i++)
    {
        T = AT[r.Next(4)];
        L = AL[r.Next(4)];
        empArray.Add(new NhanVien(T, L));
    }
    return empArray;
}
}

```

Kết quả chạy chương trình:



```

=====
Danh sach cac nhan vien:
Chi Ba, 800
Chi Ba, 700
Anh Ba, 600
Chi Hai, 600
Anh Hai, 800

Sau khi sap xep theo Ten:
Anh Ba, 600
Anh Hai, 800
Chi Ba, 800
Chi Ba, 700
Chi Hai, 600

Sau khi sap xep theo Luong:
Chi Hai, 600
Anh Ba, 600
Chi Ba, 700
Chi Ba, 800
Anh Hai, 800

```

e) Kế thừa giao diện IComparer

Lớp **NhanVien** không nhất thiết phải kế thừa giao diện **IComparable**. Thay vì vậy, việc so sánh hai đối tượng **NhanVien** được giao toàn bộ cho một trọng tài (**Referee**). Lớp **Referee** này phải kế thừa giao diện **IComparer**. Khi định nghĩa lại phương thức trọng tài sẽ truy cập đến thông tin của hai đối tượng **NhanVien** đó tùy theo tiêu chí (tên hoặc lương) để so sánh và trả về kết quả.

Đây có thể xem như trường hợp hai đối tượng **NhanVien** không biết cách tự so sánh với nhau, hoặc không biết cần dùng tiêu chí nào. Vì cách sử dụng các đối tượng **NhanVien** còn tùy thuộc vào từng ứng dụng cụ thể sử dụng lại lớp này.

Ví dụ 4.

```
using System;
```

```
using System.Collections;

class NhanVien
{
    string Ten;
    long Luong;
    public NhanVien(string T, long L)
    {
        Ten = T;
        Luong = L;
    }
    public override string ToString()
    {
        return Ten + " , " + Luong;
    }

    // Gọi ra 1 trọng tài có thể so sánh 2 nhân viên
    // Mục đích là để cung cấp tham số cho các hàm như Sort, BinarySearch của
    ArrayList
    public static Referee GetComparer()
    {
        return new Referee();
    }

    // Lớp lồng nhau (inner class) Referee dùng để thiết lập thiết lập tiêu
    chí so sánh 2 nhân viên
    public class Referee : System.Collections.IComparer
    {
        TieuChiSoSanh tieu_chi;

        //Quy định các tiêu chí so sánh 2 đối tượng
        public enum TieuChiSoSanh
        {
            Ten, Luong
        }

        // Lấy hoặc gán lại tiêu chí so sánh 2 đối tượng
        public TieuChiSoSanh TieuChi
        {
            get { return tieu_chi; }
            set { tieu_chi = value; }
        }
        // so sánh 2 đối tượng theo 1 tiêu chí đã xác lập
        public int Compare(Object lhs, Object rhs)
        {
            NhanVien r = (NhanVien)rhs;
            NhanVien l = (NhanVien)lhs;
            switch (TieuChi)
            {
                case TieuChiSoSanh.Ten:
                    return l.Ten.CompareTo(r.Ten);
                case Referee.TieuChiSoSanh.Luong:
                    return l.Luong.CompareTo(r.Luong);
                default:
                    return 0;
            }
        }
    }
}
```

```
class App
{
    static void Main(string[] args)
    {
        ArrayList empArray = TaoCacNhanVien(5);
        Console.WriteLine("=====");
        Console.WriteLine("Danh sach cac nhan vien:");
        XuatCacNhanVien(empArray);

        // Sắp xếp theo tên (mặc định)
        // empArray.Sort(); --> Lỗi không thể gọi hàm này nếu lớp NhanVien
        không kế thừa giao diện IComparable
        Console.WriteLine("\nSau khi sap xep theo Ten:");
        NhanVien.Referee rf = new NhanVien.Referee();
        empArray.Sort(rf); // Sắp xếp theo tên do tiêu chí mặc định của
        Referee
        XuatCacNhanVien(empArray);

        // Sắp xếp theo lương
        rf.TieuChi = NhanVien.Referee.TieuChiSoSanh.Luong;
        empArray.Sort(rf);
        Console.WriteLine("\nSau khi sap xep theo Luong:");
        XuatCacNhanVien(empArray);

        Console.ReadLine();
    }

    private static void XuatCacNhanVien(ArrayList empArray)
    {
        foreach (NhanVien x in empArray) Console.WriteLine(x.ToString());
    }

    private static ArrayList TaoCacNhanVien(int SoNV)
    {
        string[] AT = { "Anh Hai", "Chi Hai", "Anh Ba", "Chi Ba" };
        long[] AL = { 500, 600, 700, 800 };

        ArrayList empArray = new ArrayList();
        Random r = new Random(10);
        string T;
        long L;
        //Tạo 2 nhân viên có và Lương lấy ngẫu nhiên từ 2 mảng trên
        for (int i = 0; i < SoNV; i++)
        {
            T = AT[r.Next(4)];
            L = AL[r.Next(4)];
            empArray.Add(new NhanVien(T, L));
        }
        return empArray;
    }
}
```

Kết quả chạy chương trình:

```
file:///D:/Day 2022-2023/OOP K44/Thu
=====
Danh sach cac nhan vien:
Chi Ba , 800
Chi Ba , 700
Anh Ba , 600
Chi Hai , 600
Anh Hai , 800

Sau khi sap xep theo Ten:
Anh Ba , 600
Anh Hai , 800
Chi Ba , 800
Chi Ba , 700
Chi Hai , 600

Sau khi sap xep theo Luong:
Chi Hai , 600
Anh Ba , 600
Chi Ba , 700
Chi Ba , 800
Anh Hai , 800
```

Trong ví dụ 4 sau đây, lớp **Referee** là một lớp nội tại (inner class - lớp định nghĩa bên trong một lớp khác) của lớp **NhanVien**. Cách cài đặt lớp **Referee** như là một lớp nội tại thường được áp dụng khi có những tiêu chí so sánh cần đến các dữ liệu/thông tin lớp **NhanVien** không cho phép bên ngoài lớp truy cập (**private**, **protected**), chỉ cho phép bên ngoài chọn lựa tiêu chí so sánh. Bạn có thể cài đặt thành một lớp độc lập như thông thường nếu các tiêu chí so sánh chỉ cần đến các dữ liệu public của lớp **NhanVien**).

Bài tập 1. Sửa lại Ví dụ 4, bằng cách cài đặt lớp **Referee** thành một lớp độc lập mà không phải là inner-class của lớp **NhanVien**.

Bài tập 2. Dựa vào Ví dụ 2 hoặc Ví dụ 3 ở trên hãy đưa ra một ví dụ về việc kế thừa giao diện **IComparable** và gọi phương thức **BinarySearch**.

Bài tập 3. Dựa vào Ví dụ 4 ở trên hãy đưa ra một ví dụ về việc kế thừa giao diện **IComparer** và gọi phương thức **BinarySearch**.

Bài tập 4 (Tự suy luận hoặc tìm kiếm). Ngoài các phương thức **BinarySearch**, **Sort** còn có phương thức nào khác của **ArrayList** yêu cầu lớp của các phần tử kế thừa giao diện **IComparable** hoặc **IComparer** nữa không?

III.9.8. Câu hỏi ôn tập

1. Sự chuyên biệt hóa/tổng quát hóa mô hình hóa trong C# thông qua tính gì?
2. Khái niệm đa hình là gì? Khi nào thì cần sử dụng tính đa hình?
3. Hãy xây dựng cây phân cấp các lớp đối tượng sau: *Xe_Toyota*, *Xe_Dream*, *Xe_Spacey*, *Xe_BMW*, *Xe_Fiat*, *Xe_DuLich*, *Xe_May*, *Xe*?
4. Từ khóa **new** được sử dụng làm gì trong các lớp?
5. Một phương thức ảo trong lớp cơ sở có nhất thiết phải được phủ quyết (định nghĩa lại) trong lớp dẫn xuất hay không?
6. Lớp trừu tượng có cần thiết phải xây dựng hay không? Hãy cho một ví dụ về một lớp trừu tượng cho một số lớp.

7. Lớp Object cung cấp những phương thức nào mà các lớp khác thường xuyên kế thừa để sử dụng.
8. Phân biệt lớp cha (base class), lớp con (drived class), lớp trừu tượng và giao diện.
9. Tìm hiểu về mô hình 3 lớp trong C#.

III.9.9. Bài tập tổng hợp

1. Hãy xây dựng các lớp đối tượng trong câu hỏi 3, thiết lập các quan hệ kế thừa dựa trên cây kế thừa mà bạn xây dựng. Mỗi đối tượng chỉ cần một thuộc tính là myName để cho biết tên của nó (như Xe_Toyota thì myName là “Toi la Toyota”...). Các đối tượng có phương thức Who() cho biết giá trị myName của nó. Hãy thực thi sự đa hình trên các lớp đó. Cuối cùng tạo một lớp Tester với hàm Main() để tạo một mảng các đối tượng Xe, đưa từng đối tượng cụ thể vào mảng đối tượng Xe, sau đó cho lặp từng đối tượng trong mảng để nó tự giới thiệu tên (bằng cách gọi hàm Who() của từng đối tượng).
 2. Xây dựng các lớp đối tượng hình học như: điểm, đoạn thẳng, đường tròn, hình chữ nhật, hình vuông, tam giác, hình bình hành, hình thoi. Mỗi lớp có các thuộc tính riêng để xác định được hình vẽ biểu diễn của nó như đoạn thẳng thì có điểm đầu, điểm cuối.... Mỗi lớp thực thi một phương thức Draw() phủ quyết phương thức Draw() của lớp cơ sở gốc của các hình mà nó dẫn xuất. Hãy xây dựng lớp cơ sở của các lớp trên và thực thi đa hình với phương thức Draw(). Sau đó tạo lớp Tester cùng với hàm Main() để thử nghiệm đa hình giống như bài tập 1 ở trên.
-

PHỤ LỤC

Phụ lục A - CƠ BẢN VỀ NGÔN NGỮ C# (C - Sharp)

I. Tạo ứng dụng trong C#

Ví dụ dưới đây là một ứng dụng dòng lệnh (console application) đơn giản. Ứng dụng này giao tiếp với người dùng thông qua bàn phím, màn hình DOS và không có giao diện đồ họa người dùng, giống như các ứng dụng thường thấy trong Windows.

Ứng dụng dòng lệnh trong ví dụ sau sẽ xuất chuỗi "Hello World" ra màn hình. Thông qua ví dụ này chúng ta nắm được cấu trúc của một chương trình trong C#.

Ví dụ:

```
using System; //Khai báo thư viện
class HelloWorld //Khai báo lớp
{
    //Định nghĩa hàm Main
    static void Main( ) {

        //Xuất câu thông báo "Hello ra màn hình"
        System.Console.WriteLine("Hello World");

        //Chờ người dùng gõ một phím bất kỳ để thoát
        System.Console.ReadLine();
    }
}
```

Chương trình trên khai báo một kiểu đơn giản: lớp *HelloWorld* bằng từ khóa *class*, được bao bởi cặp dấu {}, trong đó có một phương thức (hàm) tên là *Main()*. Khi chạy chương trình, hàm *Main()* được thực thi đầu tiên nên mỗi chương trình chỉ có duy nhất một hàm *Main()*. Ngoài hàm *Main()* lớp *HelloWorld* có thể có thêm các hàm khác.

Mỗi lớp có các dữ liệu (thuộc tính) và các hành vi đặc trưng của lớp đó. Thuộc tính là các thành phần dữ liệu mà mọi đối tượng thuộc lớp đều có. Hành vi là phương thức của lớp (còn gọi là hàm thành viên), đó là các hàm thao tác trên các thành phần dữ liệu của lớp. Trong ví dụ này, lớp *HelloWorld* không có thuộc tính dữ liệu và hành vi nào (trừ hàm *Main()*).

Trong ví dụ này, ta sử dụng đối tượng *Console* để thao tác với bàn phím và màn hình. Đối tượng *Console* thuộc không gian (name space, thư viện) *System* vì vậy ta sử dụng chỉ thị *using System* ở đầu chương trình.

Để truy cập đến một thành phần của lớp hoặc của đối tượng ta dùng toán tử chấm ".". Lệnh *System.Console.WriteLine("Hello World");* có nghĩa là gọi hàm *WriteLine* của đối tượng *Console* trong thư viện *System* để xuất chuỗi "Hello World" ra màn hình.

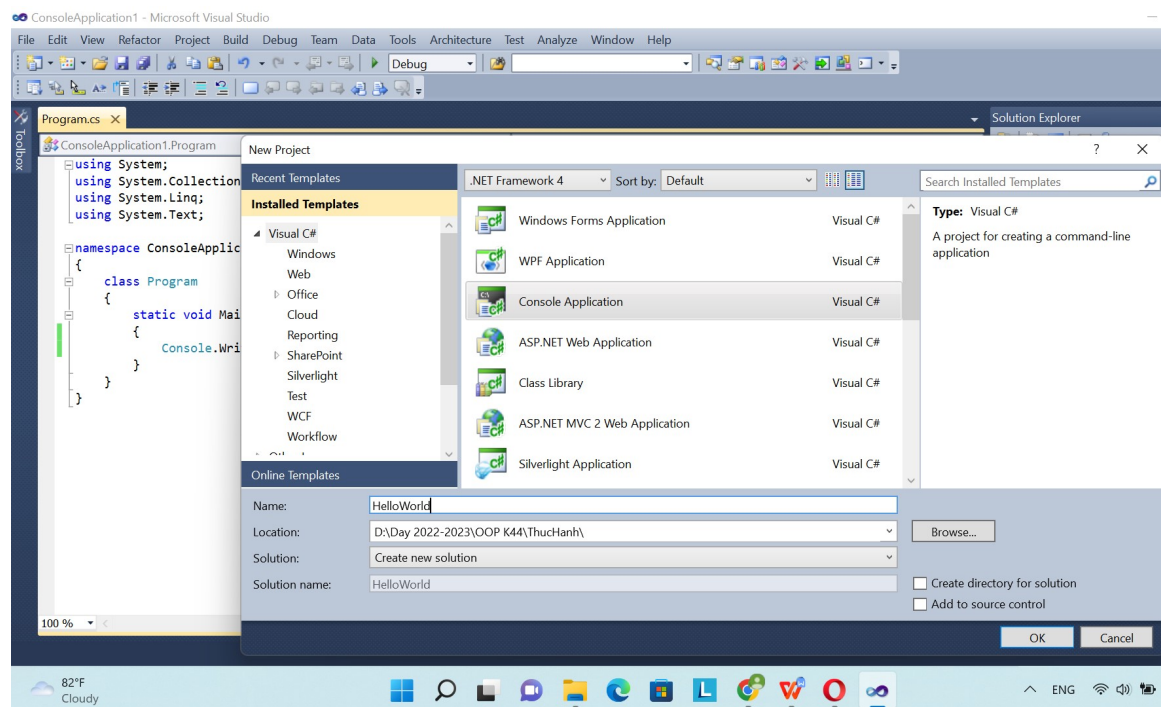
Lệnh `System.Console.ReadLine()`; thực chất dùng để nhập một chuỗi từ bàn phím. Trong trường hợp này nó có tác dụng chờ người dùng nhấn phím Enter để kết thúc chương trình.

Chú ý:

Phần 1 này trình bày các chương trình theo phương pháp lập trình thủ tục truyền thống nhằm làm quen với ngôn ngữ C# vì vậy, các ví dụ không được trình bày theo phương pháp hướng đối tượng.

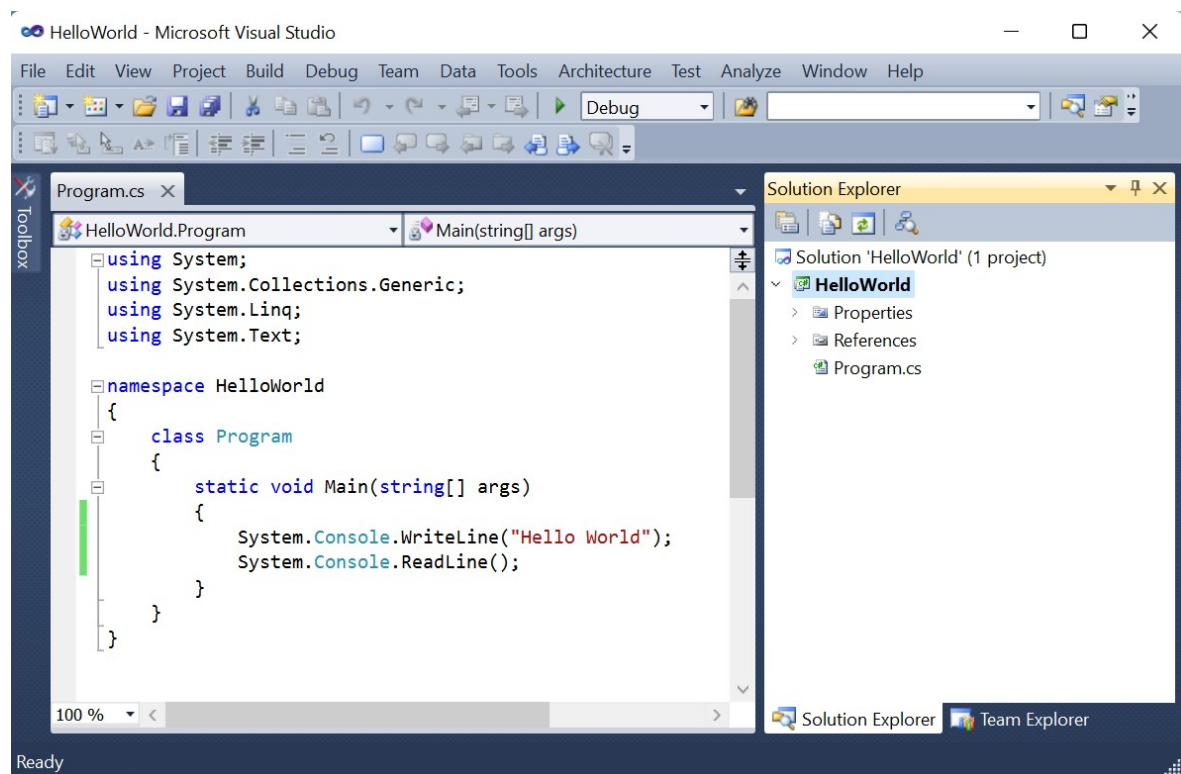
1.1. Soạn thảo chương trình “Hello World”

- Khởi động MS Visual Studio.
- Tạo ứng dụng dòng lệnh tên là Hello World qua các bước sau: **File\New\Project**. Chọn **Visual C# Project** trong ô **Project Types** và chọn **Console Application** trong ô **Templates** như hình dưới đây. Nhập vào tên dự án là **HelloWorld** vào ô **Name** và đường dẫn để lưu trữ dự án vào ô **Location** (ví dụ, **D:\Day 2022-2023\OOP K44\ThucHanh**).



Hình I-1: Tạo ứng dụng C# Console Application trong Visual Studio.

- Sau đó đưa lệnh `System.Console.WriteLine("Hello World");` vào trong phương thức `Main()` như Hình I-2.



Hình I- 2: Mã chương trình HelloWorld.

1.2. Biên dịch và chạy chương trình “Hello World”

- Biên dịch chương trình: **Ctrl+Shift+B Build→Build** hay **Build→Build**.
- Chạy chương trình và dò lỗi: **F5** hay **Debug→Start Debugging**.
- Biên dịch và chạy chương trình (không dò lỗi): **Ctrl + F5** hay **Debug→Start Without Debugging**.

II. Cơ sở của ngôn ngữ C#

Phần này sẽ trình bày các kiến thức cơ bản cho việc lập trình trong ngôn ngữ C# như:

- Các kiểu dữ liệu xây dựng sẵn như: int, bool, string...
- Hằng, cấu trúc liệt kê
- Chuỗi, mảng
- Biểu thức và câu lệnh
- Các cấu trúc điều khiển như if, switch, while, do...while, for, và foreach...

Nắm vững phần này sẽ giúp ta hiểu phương pháp lập trình hướng đối tượng một cách nhanh chóng, dễ dàng.

II.1. Các kiểu dữ liệu

Có nhiều cách phân loại kiểu dữ liệu trong C#. Ví dụ, kiểu dữ liệu có thể được phân làm 2 loại sau:

- Kiểu dựng sẵn: int, long ...
- Kiểu người dùng tạo ra: class, struct...

Tuy nhiên, người lập trình thường quan tâm tới cách phân loại sau vì nó ảnh hưởng tới cách chúng ta gán giá trị cho biến, truyền tham số cho hàm:

- **Kiểu giá trị như: int, long, char, bool, struct.** Giá trị thật sự của chúng được cấp phát trên stack. Hai biến khác nhau sẽ chứa giá trị nằm ở hai vùng nhớ khác nhau.
- **Kiểu tham chiếu như: lớp, mảng.** Địa chỉ của kiểu tham chiếu được lưu trữ trên stack nhưng dữ liệu thật sự lưu trữ trong heap. Hai biến khác nhau có thể trỏ tới cùng một vùng nhớ.

Chú ý:

Tất cả các kiểu dữ liệu xây dựng sẵn là kiểu dữ liệu giá trị ngoại trừ các đối tượng, mảng và chuỗi. Và tất cả các kiểu do người dùng định nghĩa ngoại trừ kiểu cấu trúc đều là kiểu dữ liệu tham chiếu.

II.1.1. Các kiểu xây dựng sẵn trong C#:

Ngôn ngữ C# đưa ra các kiểu dữ liệu xây dựng sẵn rất hữu dụng, phù hợp với một ngôn ngữ lập trình hiện đại.

Mỗi kiểu nguyên thủy của C# được ánh xạ đến một kiểu dữ liệu được hỗ trợ bởi hệ thống xác nhận ngôn ngữ chung (Common Language Specification: CLS) trong MS.NET. Mỗi kiểu tương ứng trong CLS này là một class. Việc ánh xạ này đảm bảo các đối tượng được tạo ra trong C# có thể được sử dụng lại một cách tương thích với các đối tượng được tạo bởi bất cứ ngôn ngữ khác được biên dịch bởi .NET, chẳng hạn VB.NET.

Kiểu trong C#	Kích thước (byte)	Kiểu tương ứng trong .Net (CLS)	Mô tả
byte	1	Byte	Không dấu 0 → 255
char	1	Char	Ký tự unicode
bool	1	Boolean	True hay false
sbyte	1	Sbyte	Có dấu (-128 → 127)
short	2	Int16	Có dấu -32.768 → 32.767

ushort	2	UInt16	Không dấu (0 → 65535)
int	4	Int32	Có dấu -2,147,483,647 → 2,147,483,647.
uint	4	UInt32	Không dấu 0 → 4,294,967,295.
float	4	Single	$\pm 1.5 * 10^{-45} \rightarrow \pm 3.4 *$
double	8	Double	$\pm 5.0 * 10^{-324} \rightarrow \pm 1.7 *$
decimal	8	Decimal	Lên đến 28 chữ số.
long	8	Int64	-9,223,372,036,854,775,808 → 9,223,372,036,854,775,807
ulong	8	UInt64	0 to 0xffffffffffffffff.

C# đòi hỏi các biến phải được khởi gán giá trị trước khi truy xuất giá trị của nó.

II.1.2. Hằng

Cú pháp:

const kiểu tên_biến = giá trị.

Ví dụ II.1.2: Khai báo hai hằng số **DoDongDac**, **DoSoi** (nhiệt độ đông đặc và nhiệt độ sôi).

```
using System;
class Values{
    const int DoDongDac = 32; // Nhiệt độ đông đặc
    const int DoSoi = 212;    //Độ sôi
    static void Main( )
    {
        Console.WriteLine("Nhiệt độ đông đặc của nước:
        {0}", DoDongDac);
        Console.WriteLine("Nhiệt độ sôi của nước: {0}",
        DoSoi);
    }
}
```

II.1.3. Kiểu liệt kê

Kiểu liệt kê là một kiểu giá trị rời rạc, bao gồm một tập các hằng có liên quan với nhau. Mỗi biến thuộc kiểu liệt kê chỉ có thể nhận giá trị là một trong các hằng đã liệt kê. Chẳng hạn, thay vì khai báo dài dòng và rời rạc như sau:

```
const int DoDongDac = 32; // Nhiệt độ đông đặc
const int DoSoi = 212;    //Độ sôi
const int HoiLanh = 60;
```

```
const int AmAp = 72;
```

ta có thể định nghĩa kiểu liệt kê có tên là `NhietDo` như sau:

```
enum NhietDo{
    DoDongDac = 32; // Nhiệt độ đông đặc
    DoSoi = 212;    //Độ sôi
    HoiLanh = 60;
    AmAp = 72;
}
```

Mỗi kiểu liệt kê có thể dựa trên một kiểu cơ sở như `int`, `short`, `long`..(trừ kiểu `char`). Mặc định là kiểu `int`.

Ví dụ II.1.3.1: Xây dựng kiểu liệt kê mô tả kích cỡ của một đối tượng nào đó dựa trên kiểu số nguyên không dấu:

```
enum KichCo: uint {
    Nho = 1;
    Vua = 2;
    Rong = 3;
}
```

Ví dụ II.1.3.2: Ví dụ minh họa dùng kiểu liệt kê để đơn giản mã chương trình:

```
using System;
enum NhietDo{
    GiaBuot = 0, DongDac = 32, AmAp = 72, NuocSoi = 212
}

class Enum{

    static void Main(string[] args) {
        Console.WriteLine("Nhiệt độ đông đặc của nước: {0}", NhietDo.DoDongDac);
        Console.WriteLine("Nhiệt độ sôi của nước: {0}", NhietDo.DoSoi);
    }
}
```

Mỗi hằng trong kiểu liệt kê tương ứng với một giá trị. Nếu chúng ta không chủ động gán giá trị cho các hằng số trong kiểu liệt kê thì giá trị mặc định của hằng số là 0 và tăng dần theo thứ tự cho các phần tử tiếp theo.

Ví dụ II.1.3.3

```
enum SomeValues{
    First,      Second, Third = 20, Fourth
}
```

Khi đó: `First = 0`, `Second = 2`, `Third = 20`, `Fourth = 21`.

II.1.4. Kiểu chuỗi

Lớp *string* (hoặc *String*) là lớp mô tả chuỗi các ký tự. Chuỗi là một mảng các ký tự nên ta có thể truy cập đến từng ký tự tương tự như cách truy cập đến một phần tử của mảng.

Ta khai báo một biến *string* sau đó gán giá trị cho biến *string* (vừa khai báo vừa khởi gán giá trị) như sau:

```
string myString = "Hello World";
```

Chú ý:

Kiểu string là kiểu bất biến (immutable), ta có thể gán (toàn bộ) một giá trị mới cho một biến kiểu string nhưng không thể thay đổi một vài ký tự trong chuỗi.

Ví dụ II.1.4.1: Có thể thực hiện các lệnh sau:

```
string S1 = "Hello World";
```

```
S1 = "how are you?";
```

Ví dụ II.1.4.2: Không thể thực hiện các lệnh sau:

```
string S1 = "Hello World";
```

```
S1[0] = ' h' ;
```

II.2. Lệnh rẽ nhánh

II.2.1. Lệnh if

Ví dụ II.2.1: Nhập một số nguyên, kiểm tra số vừa nhập là chẵn hay lẻ.

```
using System;
namespace IfExample
{
    class IF
    {
        static void Main(string[] args)
        {
            int Value;
            Console.WriteLine("Nhap mot so nguyen!");

            //Nhập một số nguyên từ bàn phím và gán cho biến Value
            Value = Int32.Parse(Console.ReadLine());
            if (Value % 2 == 0)
                Console.WriteLine("Ban nhap so chan!");
            else
                Console.WriteLine("Ban nhap so le!");
        }
    }
}
```

```
        Console.Read();  
    }  
}
```

II.2.2. Lệnh switch

Cú pháp:

```
switch (Biểu thức)  
{  
    case hằng số_1:  
        Các câu lệnh  
        Lệnh nhảy thoát khỏi case này  
    case hằng số_2:  
        Các câu lệnh  
        Lệnh nhảy thoát khỏi case này  
  
    ...  
    [default: các câu lệnh]  
}
```

Ví dụ II.2.2 Hiện một thực đơn và yêu cầu người dùng chọn một

```
using System;  
enum ThucDon:int {  
    Xoai,Oi,Coc  
}  
  
//Minh họa lệnh switch  
class Switch{  
    static void Main(string[] args)    {  
        ThucDon Chon;  
        NhapLai:  
        Console.WriteLine("Chon mot mon!");  
        Console.WriteLine("{0} - Xoai", (int)ThucDon.Xoai);  
        Console.WriteLine("{0} - Oi", (int)ThucDon.Oi);  
        Console.WriteLine("{0} - Coc", (int)ThucDon.Coc);  
        Chon=(ThucDon) int.Parse(Console.ReadLine());  
        if (Chon < 0) goto NhapLai;  
  
        switch(Chon)    {  
            case ThucDon.Xoai:  
                Console.WriteLine("Chon 1 qua  
                xoai!");  
                break;  
            case ThucDon.Oi:  
                Console.WriteLine("Chon an 1 qua oi!");  
                break;  
            case ThucDon.Coc:  
                Console.WriteLine("Chon an 1 con coc!");  
                break;  
        }  
    }  
}
```

```
        default:
            Console.WriteLine("Vui long chon lai!");
            break;
    }
    Console.WriteLine("Chuc ban ngon mieng!");
    Console.ReadLine();
}
}
```

Ghi chú:

Trong C#, nếu case trùng với biểu thức của lệnh switch rỗng thì các chương trình tự động nhảy xuống lệnh case tiếp theo cho đến khi gặp case không rỗng. Case không rỗng cần có 1 lệnh nhảy như break, goto, return... để thoát khỏi lệnh switch.

II.2.3. Lệnh goto

Ví dụ II.2.3: Xuất các số từ 0 đến 9

```
using System;
public class Tester
{
    public static int Main( )
    {
        int i = 0;
        LapLai: // nhãn
        Console.WriteLine("i: {0}",i);
        i++;
        if (i < 10) goto LapLai; // nhảy tới nhãn LapLai
        return 0;
    }
}
```

II.2.4. Lệnh lặp while

Ví dụ II.2.4: Phân tích số nguyên dương N ra thừa số nguyên tố.

```
using System;
class While{
    static void Main(string[] args)    {
        int N, M, i;
        Console.WriteLine("Nhap so nguyen duong (>1): ");
        N= int.Parse(Console.ReadLine());
        if (N <2) {
            Console.WriteLine("So khong hop le ");
            return;
        }
    }
}
```



```
    }
    string KetQua = "";
    i = 2;
    M = N;
    while(M > 1)    {
        if (M%i == 0)    {
            M = M/i;
            if (KetQua.Equals(""))KetQua += i;
            else KetQua = KetQua + "*" + i;
        }
        else i++;
    }
    Console.WriteLine("Số {0} o dang thua so nguyen
to la:{1}", N, KetQua);
    Console.ReadLine();
}
}
```

II.2.5. Lệnh do...while

Cú pháp:

do <lệnh> while <biểu_thức>.

Vòng lặp **do ...while** thực hiện ít nhất 1 lần.

Ví dụ II.2.5: Kết quả của đoạn lệnh sau là gì?

```
using System;
public class Tester{
    public static int Main( )    {
        int i = 11;
        do    {
            Console.WriteLine("i: {0}",i);
            i++;
        } while (i < 10);
        return 0;
    }
}
```

II.2.6. Lệnh for

Cú pháp:

for (khởi tạo; điều kiện dừng; lặp) lệnh;

Ví dụ II.2.6: Kiểm tra số nguyên tố

```
using System;
class NguyenTo{
    static void Main(string[] args)    {
```

```
int N, i;
Console.WriteLine("Nhap so nguyen duong (>1): ");
N= int.Parse(Console.ReadLine());
if (N <2) {
    Console.WriteLine("So khong hop le ");
    return;
}
bool KetQua;
KetQua = true;
for ( i = 2; i<= Math.Sqrt(N); i++){
    if (N%i==0) {
        KetQua = false;
        break;
    }
}
if (KetQua)
    Console.WriteLine("{0} la so nguyen to",N);
else
    Console.WriteLine("{0} khong la so nguyen to",N);

Console.ReadLine();
}
}
```

II.2.7. Lệnh foreach

Vòng lặp **foreach** cho phép tạo vòng lặp duyệt qua một tập hợp hay một mảng. Câu lệnh **foreach** có cú pháp chung như sau:

***foreach (<kiểu thành phần> <tên truy cập thành phần > in < tên tập hợp>)**
<Các câu lệnh thực hiện>*

Ví dụ II.2.7: Xuất các kí tự trong chuỗi.

```
using System;
using System.Collections.Generic;
using System.Text;

public class UsingForeach {
    public static int Main()    {
        string S = "He no";
        string[] MonAn;
        MonAn = new string[3] {"Ga", "Vit", "Ngan"};

        foreach (char item in S)    {
            Console.Write("{0} ", item);
        }
        Console.WriteLine();
        foreach (string Si in MonAn)
            Console.Write("{0} \n", Si);

        System.Console.Read();
        return 0;
    }
}
```

```
    }  
}
```

II.2.8. Lệnh continue và break

Lệnh *continue* được dùng trong vòng lặp (for, while, do... while), cho phép bỏ qua các lệnh phía sau nó và quay lại đầu vòng lặp.

Lệnh *break* cho phép thoát ra khỏi vòng lặp gần nó nhất;

Ví dụ II.2.8: Chương trình cho phép liên tục nhập các số nguyên dương để kiểm tra chúng có phải là số nguyên tố hay không cho đến khi nhập số 0 hoặc số 1. Nếu người dùng nhập số âm thì cho phép nhập lại.

```
public class Break_Continue    {  
    public static bool laSoNguyenTo(int M)    {  
        if (M < 2) return false;  
        for (int i = 2; i <= Math.Sqrt(M); i++)  
            if (M % i == 0) return false;  
        return true;  
    }  
  
    public static void Main() {  
        while (true) {  
            Console.WriteLine("Nhap 1 so nguyên dương để kiểm  
tra tính chất nguyên tố");  
            Console.WriteLine("Nhap số 0 hoặc 1 để dừng CT!");  
            Console.WriteLine("Nhap lại nếu nhập số âm");  
  
            int X = Int16.Parse(Console.ReadLine());  
            if (X < 0)    {  
                Console.WriteLine("Nhap lại số nguyên  
dương!\n");  
                continue;  
            }  
            else if (X == 0 || X == 1)    {  
                Console.WriteLine("Kết thúc CT\n");  
                break;  
            }  
  
            //Thu hiện việc hủy bỏ dữ liệu ghi không thành  
cong  
            bool kq = laSoNguyenTo(X);  
            if (kq) Console.WriteLine("{0} là số nguyên  
to!\n", X);  
            else Console.WriteLine("{0} không là số nguyên  
to!\n", X);  
        }  
    }  
}
```

II.3. Mảng

Mảng thuộc loại dữ liệu tham chiếu (dữ liệu thực sự được cấp phát trong Heap).

II.3.1. Mảng một chiều

- Cú pháp khai báo mảng 1 chiều:

Kiểu [] Ten_bien; //Lúc này Ten_bien == null

- Cú pháp cấp phát cho mảng bằng từ khóa new:

Ten_bien = new Kiểu [Kích_Thước];

Ví dụ II.3.1: Nhập một mảng số nguyên, sắp xếp và xuất ra màn hình.

```
using System;
class Array
{
    public static void NhapMang(int[] a, uint n) {
        for ( int i = 0; i < n; i++){
            Console.WriteLine("Nhap phan tu thu {0}",i);
            a[i] = Int32.Parse(Console.ReadLine());
        }
    }
    public static void XuatMang(int[] a, uint n) {
        for ( int i = 0; i < n; i++)
            Console.Write("{0} ",a[i]);
    }
    public static void SapXep(int[] a, uint n){
        int i, j, temp;
        for ( i = 0; i < n-1; i++) {
            for ( j = i+1; j < n; j++) {
                if (a[i]>a[j]) {
                    temp=a[i];
                    a[i]=a[j];
                    a[j]=temp;
                }
            }
        }
    }

    public static void Main() {
        Console.WriteLine("Nhap kích thước mảng: ");
        uint n = uint.Parse(Console.ReadLine());
        int[] A;
        A = new int[n];
        NhapMang(A,n);
    }
}
```

```

        Console.WriteLine("Mang vua nhap");
        XuatMang(A,n);

        SapXep(A,n);
        Console.WriteLine("Mang sau khi sap xep");
        XuatMang(A,n);
        Console.ReadLine();
    }
}

```

II.3.2. Mảng nhiều chiều

- Cú pháp khai báo mảng 2 chiều:

Kiểu [][] Ten_bien;

Vì mảng 2 chiều là mảng mà mỗi phần tử lại là một mảng con 1 chiều nên để cấp phát đủ vùng nhớ chứa các phần tử (đủ số dòng, số cột) ta cần lần lượt thực hiện các bước sau:

1. Cấp phát mảng trỏ đến các dòng
2. Lần lượt cấp phát cho các mảng con cho từng dòng.

Cú pháp của hai bước trên như sau:

- Cấp phát một mảng các mảng (Cấp phát mảng trỏ đến các dòng):

Ten_bien = new Kiểu [Số_dòng][];

- Cấp phát cho từng mảng con thứ i (cấp phát cho các mảng con cho từng dòng):

Ten_bien [i] = new Kiểu [kích_thước_của_dòng_thứ_i];

Ví dụ II.3.2: Nhập, xuất ma trận số thực.

```

using System;
class Matrix {
    public static void NhapMaTran(float[][][] a,int n,uint m)
    {
        for ( int i = 0; i < n; i++) {
            for (int j = 0; j < m; j++) {
                Console.Write("Nhap phan tu thu
[{0},{1}]",i,j);
                a[i][j] = float.Parse(Console.ReadLine());
            }
        }
    }
}

```

```
public static void XuatMaTran(float[][]a, int n, int m)
{
    for (int i = 0; i < n; i++) {
        for (j = 0; j < m; j++)
            Console.Write("\t{0}", a[i][j]);
        Console.WriteLine();
    }
}

public static void Main() {
    uint n, m;
    Console.WriteLine("Nhap so dong cua ma tran: ");
    n = uint.Parse(Console.ReadLine());
    Console.WriteLine("Nhap so cot cua ma tran: ");
    m = uint.Parse(Console.ReadLine());

    float[][] A; //Khai báo ma trận
    //Cấp phát số dòng (n)
    A = new float[n][];

    //Cấp phát số cột (m)
    for(int i = 0; i < n; i++) A[i] = new float[m];

    NhapMaTran(A,n, m);

    Console.WriteLine("Ma tran vua nhap");
    XuatMaTran(A,n,m);
    Console.ReadLine();
}
}
```

II.3.3. Một số ví dụ khác về mảng nhiều chiều

Sau đây là một số ví dụ về mảng nhiều chiều:

- Khai báo mảng 3 chiều kiểu số nguyên với kích thước mỗi chiều là 4, 2 và 3:

```
int[, ,] myArray = new int [4,2,3];
```

- Khai báo mảng 2 chiều, cấp phát và khởi gán giá trị cho mảng thành ma trận 4 dòng 2 cột:

```
int[,] myArray = new int[,] { {1,2}, {3,4}, {5,6},
                               {7,8} };
```

hoặc

```
int[,] myArray = {{1,2}, {3,4}, {5,6}, {7,8}};
```

- Gán giá trị cho một phần tử trong mảng 2 chiều:

```
myArray[2,1] = 25;
```

II.3.4. Truyền tham số cho hàm

Khi định nghĩa một hàm, ta có thể quy định cách truyền tham số cho hàm theo kiểu tham trị hay kiểu tham chiếu.

II.3.5. Truyền tham số cho hàm theo kiểu tham trị

Khi truyền tham số cho hàm theo kiểu tham trị, sẽ xảy ra việc sao chép giá trị từ đối số (tức là giá trị hoặc biến tham số thực khi gọi hàm) sang tham số (tức là biến tham số hình thức khi định nghĩa hàm). Có thể nói một cách khác, khi gọi thực hiện hàm, đối số và tham số là hai biến khác nhau nhưng có giá trị giống nhau.

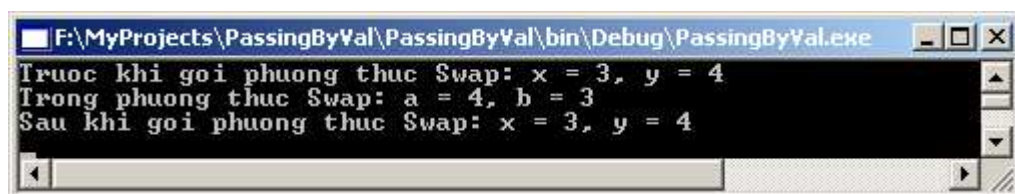
Ví dụ:

Xét phương thức **Swap** trong ví dụ sau đây, **a** và **b** là hai tham số được truyền theo dạng tham trị. Trong hàm **Main**, khi ta gọi thực hiện lệnh **Swap(x,y)** thì sẽ xảy ra sự sao chép giá trị từ các đối số **x** và **y** sang cho các biến tham số **a** và **b**. Mọi sự thay đổi chỉ diễn ra trong thân phương thức này chỉ ảnh hưởng đến các tham số **a** và **b** mà không ảnh hưởng đến các đối số **x** và **y** nên sau khi kết thực hàm, giá trị của **x** và **y** vẫn như cũ (**x** = 3 và **y** = 4).

```
using System;
class PassingByVal {
    static void Swap(int a, int b){
        int Temp = a;
        a = b;
        b = Temp;
        Console.WriteLine("Trong phuong thuc Swap: a = {0}, b = {1}", a, b);
    }

    static void Main(string[] args) {
        int x = 3, y = 4;
        Console.WriteLine("Truoc khi goi phuong thuc Swap: x = {0}, y = {1}", x, y);
        Swap(x,y);
        Console.WriteLine("Sau khi goi phuong thuc Swap: x = {0}, y = {1}", x, y);
        Console.ReadLine();
    }
}
```

Kết quả chương trình:



II.3.6. Truyền tham số cho hàm theo kiểu tham chiếu

Khi truyền tham số cho hàm theo kiểu tham chiếu thì biến tham số cũng chính là biến đối số đang được dùng để gọi thực hiện hàm. Do vậy, những câu lệnh bên trong hàm làm thay đổi biến tham số cũng sẽ làm thay đổi biến đối số. Khi kết thúc hàm những thay đổi đó vẫn còn hiệu lực đối với đối số.

Khi định nghĩa hàm, ta đặt từ khóa **ref** đứng trước biến tham số cần truyền theo kiểu tham chiếu. Khi gọi hàm, biến đối số tương ứng phải khác **null**.

Từ khóa **out** để truyền đối số cho phương thức theo kiểu tham chiếu mà không cần khởi gán giá trị đầu cho đối số. Trong hàm phải có lệnh gán giá trị cho tham chiếu này nếu ban đầu nó bằng **null**. Biến tham chiếu với từ khóa **out** thường được dùng khi hàm có nhiều giá trị trả về.

Đối với những dữ liệu kiểu giá trị (**int**, **long**, **float**, **char**,...), muốn thay đổi giá trị của chúng thông qua việc truyền tham số cho hàm, phương thức ta phải truyền theo kiểu tham chiếu một cách tường minh bằng từ khóa **ref** hoặc **out**.

Ví dụ:

Để phương thức **Swap** có thể hoán vị hai biến số nguyên như mong muốn ta phải sửa lại tham số *a*, *b* theo kiểu tham chiếu như sau:

```
static void Swap(ref int a, ref int b)
```

Khi đó ta gọi phương thức **Swap** với hai đối số *x*, *y* theo cú pháp:

```
Swap(ref x, ref y);
```

Chương trình hoàn chỉnh:

```
using System;
class PassingByVal {
    static void Swap(ref int a, ref int b) {
        int Temp = a;
        a = b;
        b = Temp;
        Console.WriteLine("Trong phuong thuc Swap: a = {0}, b = {1}", a, b);
    }

    static void Main(string[] args) {
        int x = 3, y = 4;
        Console.WriteLine("Truoc khi goi phuong thuc Swap: x = {0}, y = {1}", x, y);
        Swap(ref x, ref y);
        Console.WriteLine("Sau khi goi phuong thuc Swap: x = {0}, y = {1}", x, y);
        Console.ReadLine();
    }
}
```

Kết quả chương trình:


```
file:///D:/Day 2022-2023/OOP K44/ThucHanh/StaticMembers/bin/Debug/StaticMembers.EXE
Truoc khi goi phuong thuc Swap: x = 3, y = 4
Trong phuong thuc Swap: a = 4, b = 3
Sau khi goi phuong thuc Swap: x = 4, y = 3
```

II.3.7. Truyền tham số cho hàm với tham số là biến kiểu tham chiếu

Nhắc lại, về thực chất, các biến thuộc kiểu dữ liệu tham chiếu như đối tượng, mảng, chuỗi là các số nguyên lưu địa chỉ của vùng nhớ thực sự mà chúng ta môn thao tác.

Khi dùng dữ liệu kiểu tham chiếu như trên làm tham số truyền theo kiểu tham trị cho hàm (không có từ khóa *ref* hoặc *out*) thì cơ chế thực hiện tương tự như mục II.3.5. Tuy nhiên, ta có thể làm thay đổi 1 phần dữ liệu của vùng nhớ mà loại biến tham chiếu này trở tới. Những câu lệnh làm các biến tham số này trở tới vùng nhớ khác chỉ có tác dụng tức thời trong khi thực hiện hàm. Khi hàm kết thúc, các biến tham chiếu này trở tới vùng nhớ cũ. Nếu muốn thay đổi vùng nhớ mà chúng trở tới ta phải truyền loại tham số theo kiểu tham chiếu bằng cách dùng từ khóa *ref* hoặc *out* một cách tường minh.

Ví dụ:

Ví dụ sau đây minh họa việc truyền tham số là tham chiếu (*myArray*) cho hàm *Change* theo kiểu tham trị. Đối số *myArray* là một dữ liệu kiểu tham chiếu, nó trở tới một mảng các số nguyên được cấp phát động (trong hàm *Main*). Khi truyền *myArray* theo kiểu tham trị cho hàm *Change*, ta chỉ có thể làm thay đổi các phần tử của mảng này. Trong hàm *Change*, câu lệnh cấp phát lại vùng nhớ khác cho *myArray* chỉ có tác dụng trong hàm này. Khi kết thúc hàm biến *myArray* không trở tới vùng nhớ mới.

```
using System;
```

```
class PassingRefByVal
{
    static void Change(int[] arr)
    {
        // Lệnh nay thay lam thay doi gia tri cua arr.
        arr[0] = 888;
        // Cap phat lai arr, lam thay doi dia chi cua arr.
        // Lệnh nay chỉ có tác dụng cục bộ. Sau khi gọi hàm Change,
        // gia tri cua arr[0] vẫn không thay đổi.
        arr = new int[5] { -3, -1, -2, -3, -4 };
        Console.WriteLine("Trong phuong thuc Change, phan
            tu dau tien la: {0}", arr[0]);
    }

    public static void Main()
    {
        int[] myArray = { 1, 4, 5 };
        Console.WriteLine("Trong ham Main, truoc khi goi
            phuong thuc Change, myArray [0] = {0}", myArray
                [0]);
    }
}
```

```
Change(myArray) ;  
Console.WriteLine("Trong ham Main, sau khi gọi  
phương thức Change, myArray [0] = {0}", myArray  
[0]);  
Console.ReadLine();  
}  
}
```

Kết quả chương trình cho thấy: trong hàm **Change()**, ta chỉ có thể thay đổi giá trị của vùng nhớ do *myArray* trỏ tới bằng lệnh:

```
arr[0] = 888;
```

mà không thay đổi được bản thân tham chiếu *myArray* bằng lệnh cấp phát lại:

```
arr = new int[5] {-3, -1, -2, -3, -4};
```

Vì nếu thay đổi được tham chiếu *myArray* thì sau khi gọi phương thức *Change* giá trị của phần tử đầu tiên trong mảng phải bằng -3.



Nếu ta khai báo lại nguyên mẫu của phương thức *Change* như sau:

```
static void Change(ref int[] arr)
```

và trong hàm *Main()* ta thay câu lệnh **Change(myArray)** ; bằng câu lệnh:

```
Change(ref myArray);
```

thì mọi thao tác làm thay đổi giá trị của vùng nhớ trong **heap** do *myArray* trỏ tới hoặc mọi thao tác cấp phát lại vùng nhớ cho *myArray* thực hiện trong phương thức này sẽ làm thay đổi mảng gốc được cấp phát trong hàm *Main()*.

Kết quả chạy chương trình:



II.4. Không gian tên (namespace)

Có thể hiểu một không gian tên như là một thư viện. Sử dụng không gian tên giúp ta tổ chức mã chương trình tốt hơn, tránh trường hợp hai lớp trùng tên khi sử dụng các thư viện khác nhau. Ngoài ra, không gian tên được xem như là tập hợp các lớp đối tượng, và cung cấp duy nhất các định danh cho các kiểu dữ liệu và được đặt trong một cấu trúc phân cấp. Việc sử dụng không gian tên trong lập trình là một thói quen tốt, bởi vì công việc này chính là cách lưu các mã nguồn để sử dụng về

sau. Ngoài thư viện (namespace) do MS.NET và các hãng thứ ba cung cấp, ta có thể tạo riêng cho mình các không gian tên.

C# đưa ra từ khóa **using** để khai báo sử dụng không gian tên trong chương trình:

using < Tên namespace >

Trong một không gian tên ta có thể định nghĩa nhiều lớp và không gian tên .

Để tạo một không gian tên ta dùng cú pháp sau:

```
namespace <Tên namespace>
{
    < Định nghĩa lớp A>
    < Định nghĩa lớp B >
    .....
}
```

Ví dụ II.4.1 : Định nghĩa lớp Tester trong namespace Programming_C_Sharp.

```
namespace Programming_C_Sharp
{
    using System;
    public class Tester
    {
        public static int Main( )
        {
            for (int i=0;i<10;i++)
            {
                Console.WriteLine("i: {0}",i);
            }
            return 0;
        }
    }
}
```

Ví dụ II.4.2: Khai báo không gian tên lồng nhau:

```
namespace Programming_C_Sharp
{
    namespace Programming_C_Sharp_Test1
    {
        using System;
        public class Tester
        {
            public static int Main( )
```

```
        {
            for (int i=0;i<10;i++)
            {
                Console.WriteLine("i: {0}",i);
            }
            return 0;
        }
    }
}

namespace Programming_C_Sharp_Test2
{
    public class Tester
    {
        public static int Main( )
        {
            for (int i=0;i<10;i++)
            {
                Console.WriteLine("i: {0}",i);
            }
            return 0;
        }
    }
}
}
```

Phụ lục B - BIỆT LỆ (NGOẠI LỆ)

Biệt lệ (exception, còn gọi là ngoại lệ) là các dạng lỗi gặp phải khi chạy chương trình (lúc biên dịch chương trình không phát hiện được). Thường là do người dùng gây ra lúc chạy chương trình. Ví dụ, chương trình yêu cầu nhập một số nguyên nhưng người dùng lại nhập một chuỗi.

```
using System;
class Class1
{
    static void Main(string[] args)
    {
        int x=0;
        Console.WriteLine("Nhap mot so nguyen");
        x = int.Parse(Console.ReadLine());
        Console.WriteLine("So nguyen vua nhap {0}",x);
        Console.ReadLine();
    }
}
```

Khi chạy chương trình trên, nếu ta (vô tình) nhập một dữ liệu không phải là số nguyên (chẳng hạn nhập ký tự 'r'), chương trình sẽ dừng và báo lỗi runtime sau:

An unhandled exception of type 'System.FormatException' occurred in mscorlib.dll

Additional information: Input string was not in a correct format.

I. Bắt ngoại lệ

Việc bắt và xử lý biệt lệ nhằm giúp chương trình có tính dung thứ lỗi cao hơn:

- ✓ Cho phép vẫn chạy chương trình đối với những lỗi không quá quan trọng. Chẳng hạn khi nhập một giá trị nguyên, người dùng vô tình nhập một ký tự, khi đó không nhất thiết phải dừng chương trình mà chỉ thông báo lỗi và cho phép người dùng nhập lại.
- ✓ Để chương trình thân thiện hơn đối với người sử dụng. Thông báo lỗi cụ thể, thay vì dạng thông báo lỗi tiếng Anh mang tính kỹ thuật khó hiểu của hệ thống.

Việc bắt ngoại lệ được thực hiện thông qua khối *try {} catch {}* như sau:

```
try {  
    Các câu lệnh có thể gây ra biệt lệ.  
}  
catch (Loại_biệt_lệ_1 E1 ) {  
    các câu lệnh xử lý biệt lệ 1  
}  
[...  
catch (Loại_biệt_lệ_1 En ) {  
    các câu lệnh xử lý biệt lệ n  
}  
finally {  
    các câu lệnh xử lý cần thực hiện cho dù có biệt lệ  
    hay không  
}  
]
```

Chú thích:

try: Các câu lệnh có thể gây ra biệt lệ cần phải đặt trong phần *try*

catch (Loại_biệt_lệ_1 E1): Ta cần lập trình khối lệnh xử lý biệt lệ thuộc loại *Loại_biệt_lệ_1* tại phần này. Biến *E1* chứa thông tin cụ thể của biệt lệ.

finally: Một khối lệnh trong phần *finally* luôn luôn được thực thi dù đoạn mã có gây ra biệt lệ hay không. Ví dụ, nếu bạn mở một file để xử lý trong phần *try*, nó cần phải được đóng lại trong phần *finally*, nếu không những câu lệnh về sau có thể gây ra một ngoại lệ.

Để chương trình có tính dung thứ lỗi (vì đây có thể là lỗi vô tình của người sử dụng) ta cần viết lại như sau để cho người dùng nhập lại:

```
using System;
class Class1{
    static void Main(string[] args) {
        int x=0;
        Console.WriteLine("Nhap mot so nguyen");
        NHAPLAI:
        try {
            x = int.Parse(Console.ReadLine());
        }
        //catch(System.Exception e)
        catch(System.FormatException e) {
            Console.WriteLine("Loi : " + e.ToString());
            Console.WriteLine("Hay nhap lai 1 so nguyen!");
            goto NHAPLAI;
        }
        Console.WriteLine("So nguyen vua nhap {0}",x);
        Console.ReadLine();
    }
}
```

Vì đoạn mã

```
x = int.Parse(Console.ReadLine());
```

có thể gây ra biệt lệ

```
System.FormatException
```

nên ta đặt nó trong khối *try* và khối *catch* bắt biệt lệ này. Sau đó xuất thông báo lỗi, nhưng không dừng chương trình mà cho phép nhập lại bằng lệnh nhảy tới nhãn **NHAPLAI**:

```
goto NHAPLAI;
```

Vì mọi loại biệt lệ đều dẫn xuất từ *System.Exception* nên ta bất cứ biệt lệ cũng là một biệt lệ dạng *System.Exception*. Do vậy, nếu ta không biết loại biệt lệ là gì ta thay lệnh

```
catch(FormatException e)
```

bằng lệnh:

```
catch(Exception e)
```

Nếu muốn bắt mọi ngoại lệ nhưng không thông báo lỗi ta có thể sử dụng khối *catch* rỗng như sau:

```
catch {
    Console.WriteLine("Hay nhap lai 1 so nguyen!");
    goto NHAPLAI;
}
```

Khi đó chương trình cho phép nhập lại nhưng không thông báo cho người dùng lỗi là gì.

Ví dụ 2: Bắt nhiều ngoại lệ

```
using System;
class Class1{
    static void Main(string[] args){
        byte x = 0;
        NHAPLAI:
        Console.WriteLine("Nhap mot so nguyen");
        try {
            x = byte.Parse(Console.ReadLine());
        }
        //catch(System.Exception e)
        catch(FormatException e1) {
            Console.WriteLine("Loi : " +e1.ToString());
            Console.WriteLine("Hay nhap lai so nguyen");
            goto NHAPLAI;
        }
        catch(OverflowException e2) {
            Console.WriteLine("Loi : " +e2.ToString());
            Console.WriteLine("Chi duoc nhap so nho trong
khoang [0..256]. Hay nhap lai");
            goto NHAPLAI;
        }
        Console.WriteLine("So nguyen vua nhap {0}",x);
        Console.ReadLine();
    }
}
```

Khi một ngoại lệ phát sinh, chương trình sẽ nhảy ngay tới khối *catch* gần nhất có thể bắt ngoại lệ hoặc dừng nếu không có khối *catch* nào có thể bắt ngoại lệ này.

II. Ném ra biệt lệ

Để báo động sự bất thường của chương trình.

Cú pháp:

throw [biểu thức tạo biệt lệ];

Sau khi ném ra một ngoại lệ, các đoạn lệnh sau lệnh **throw** sẽ bị bỏ qua. Chương trình thực hiện việc bắt ngoại lệ hoặc dừng.

III. Khối finally

Khi đặt khối *finally* sau các khối *catch* thì cho dù có biệt lệ hay không chương trình vẫn không dừng mà sẽ thực hiện khối *finally*. Nếu bắt được ngoại lệ thì chương trình sẽ thực hiện khối *catch* tương ứng trước khi thực hiện khối *finally*.

Hãy thử thêm đoạn lệnh sau vào ví dụ trên.

```
finally {
    Console.WriteLine("So nguyen vua nhap {0}",x);
}
```

IV. Một số loại ngoại lệ:

- ***System.OutOfMemoryException***: Lỗi không thể cấp phát bộ nhớ.
- ***System.StackOverflowException***: Lỗi tràn stack. Thường là do gọi đệ qui quá sâu hoặc gọi đệ qui bị lặp vô tận.
- ***System.NullReferenceException***: Lỗi xảy ra khi truy cập tới một tham chiếu trở tới null trong khi cần một tham chiếu trở tới một đối tượng thực sự hiện hữu.
- ***System.TypeInitializationException***: Hàm constructor ném ra một ngoại lệ nhưng không có ngoại lệ catch nào bắt ngoại lệ này.
- ***System.InvalidCastException***: Xảy ra khi không thể thực hiện việc ép kiểu tường minh từ một kiểu cơ sở hoặc một giao diện sang một kiểu dẫn xuất.
- ***System.ArrayTypeMismatchException***: Kiểu của giá trị cần lưu vào mảng không hợp kiểu với kiểu của mảng.
- ***System.IndexOutOfRangeException***: Truy cập ngoài mảng.
- ***System.MulticastNotSupportedException***: Lỗi liên quan tới việc không thể kết hợp 2 delegate không null vì kiểu trả về của delegate không phải là void.
- ***System.ArithmeticException***: Lỗi số học. Chẳng hạn chia cho 0, tràn dữ liệu.
- ***System.DivideByZeroException***: Lỗi chia cho 0
- ***System.OverflowException***: Tràn dữ liệu. Chẳng hạn gán dữ liệu quá lớn cho một biến kiểu byte.
- ...

Ví dụ:

Nên bắt biệt lệ cụ thể trước, biệt lệ tổng quát

```
using System;
public class Test {
    public static void Main( ) {
        Test t = new Test( );
        t.CanAChiaB(4,-5 );
        Console.ReadLine();
    }
    public void CanAChiaB(int a,int b) {
        try {
            if (b == 0)
                throw new DivideByZeroException( );
            if (a*b<= 0)
                throw new ArithmeticException( );
            else
            {
                double kq = Math.Sqrt(a/b);
                Console.WriteLine ("Ket qua = {0}",kq );
            }
        }
        // Bat ngoai le cu the truoc
    }
}
```



```
        catch (DivideByZeroException) {
            Console.WriteLine("Loi chia cho 0!");
        }
        //Bat ngoai le tong quat sau;
        catch (ArithmeticException) {
            Console.WriteLine("Co loi so hoc gi gi do!");
        }
    }
}
```

V. Chỉ thị checked and unchecked

unchecked statements specify the overflow-checking context for integral-type arithmetic operations and conversions, as the following example shows:

```
uint a = uint.MaxValue;
unchecked
{
    Console.WriteLine(a + 1); // output: 0
}
try
{
    checked
    {
        Console.WriteLine(a + 1);
    }
}
catch (OverflowException e)
{
    Console.WriteLine(e.Message); // output: Arithmetic operation
    resulted in an overflow.
}
```

When integer arithmetic overflow occurs, the overflow-checking context defines what happens as follows:

- In a checked context, a `System.OverflowException` is thrown; if overflow happens in a constant expression, a compile-time error occurs.
- In an unchecked context, the operation result is truncated by discarding any high-order bits that don't fit in the destination type. For example, in the case of addition it wraps from the maximum value to the minimum value, as the preceding example shows.

Lưu ý

The behavior of user-defined operators and conversions in the case of the overflow of the corresponding result type can differ from the one described in the previous paragraph. In particular, **user-defined checked operators** might not throw an exception in a checked context.

For more information, see the Arithmetic overflow and division by zero and User-defined checked operators sections of the Arithmetic operators article.

To specify the overflow-checking context for an expression, you can also use the `checked` and `unchecked` operators, as the following example shows:

```
C#Sao chépChạy
double a = double.MaxValue;
```

```
int b = unchecked((int)a);
Console.WriteLine(b); // output: -2147483648
try
{
    b = checked((int)a);
}
catch (OverflowException e)
{
    Console.WriteLine(e.Message); // output: Arithmetic operation
    resulted in an overflow.
}
```

The `checked` and `unchecked` statements and operators only affect the overflow-checking context for those operations that are textually inside the statement block or operator's parentheses, as the following example shows:

C#Sao chépChạy

```
int Multiply(int a, int b) => a * b;
int factor = 2;
try
{
    checked
    {
        Console.WriteLine(Multiply(factor, int.MaxValue)); // output: -2
    }
}
catch (OverflowException e)
{
    Console.WriteLine(e.Message);
}
try
{
    checked
    {
        Console.WriteLine(Multiply(factor, factor * int.MaxValue));
    }
}
catch (OverflowException e)
{
    Console.WriteLine(e.Message); // output: Arithmetic operation
    resulted in an overflow.
}
```

At the preceding example, the first invocation of the `Multiply` local function shows that the `checked` statement doesn't affect the overflow-checking context within the `Multiply` function as no exception is thrown. At the second invocation of the `Multiply` function, the expression that calculates the second argument of the function is evaluated in a `checked` context and results in an exception as it's textually inside the block of the `checked` statement.

TÀI LIỆU THAM KHẢO

- 1) Phạm Hữu Khang, *C# 2005 cơ bản*, Nxb Lao Động Xã Hội, 2006.
- 2) Phạm Hữu Khang, *C# 2005 Tập 2-Lập trình Windows Form*, Nxb Lao Động Xã Hội, 2006.
- 3) Jesse Liberty, *Programming C#, 2nd Edition*, tr 1-320 ,Nxb OReilly.
- 4) Ben Albahari, *CSharp Essentials, 2nd Edition*, tr 1-88, Nxb OReilly.
- 5) <https://viettuts.vn/csharp/cai-dat-moi-truong-csharp>
- 6) Windows Form: <https://freetuts.net/hoc-csharp/c-sharp-windows-form>