

C# 7 – Thừa kế và Đa hình

Giảng viên: **ThS. Lê Thiện Nhật Quang**

Email: quangln.dotnet.vn@gmail.com

Website: <http://dotnet.edu.vn>

Điện thoại: **0868.917.786**

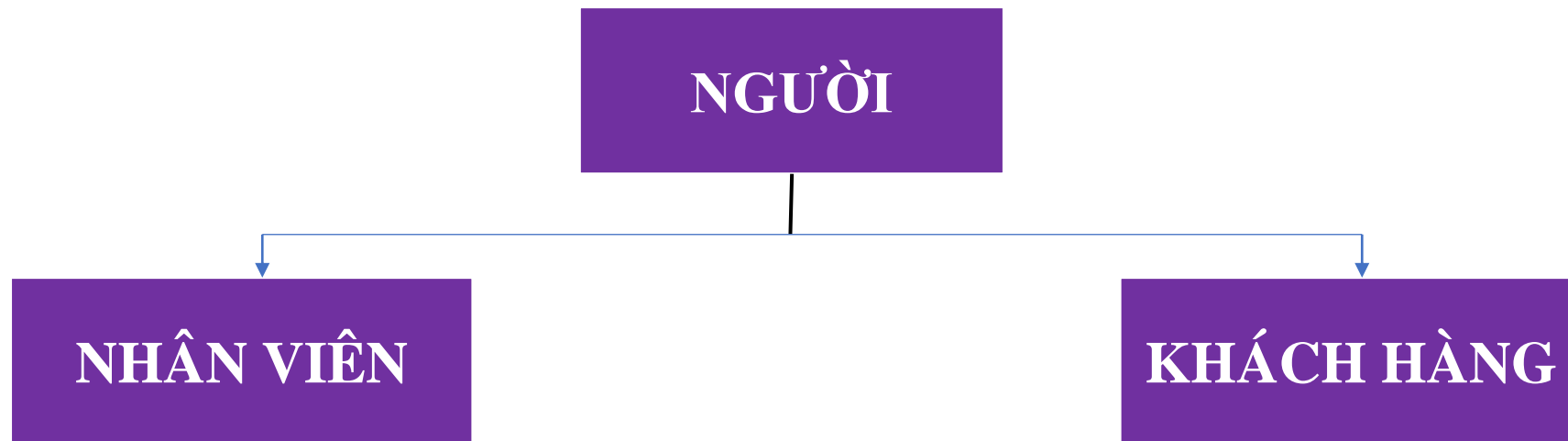


MỤC TIÊU

- Tìm hiểu về thừa kế - inheritance
- Giải thích phương thức ghi đè - overriding
- Tìm hiểu lớp sealed
- Giải thích đa hình - polymorphism

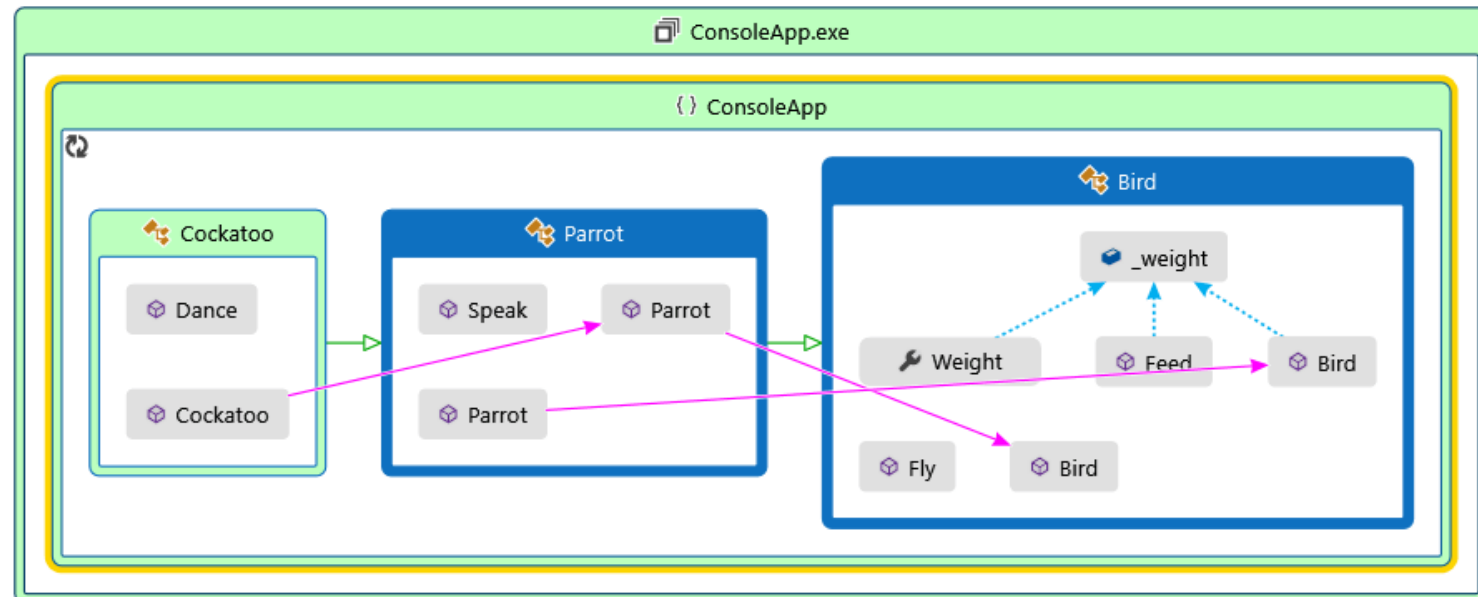
1.1. SỰ PHÂN CẤP THỪA KẾ

- ❑ Các lớp trong C# tồn tại trong một hệ thống thứ bậc phân cấp, gọi là cây thừa kế
- ❑ Lớp bậc trên gọi là lớp cha (base_class) trong khi các lớp bậc dưới gọi là lớp con (derived_class)
- ❑ Trong C# một lớp chỉ có một lớp cha duy nhất (đơn thừa kế)

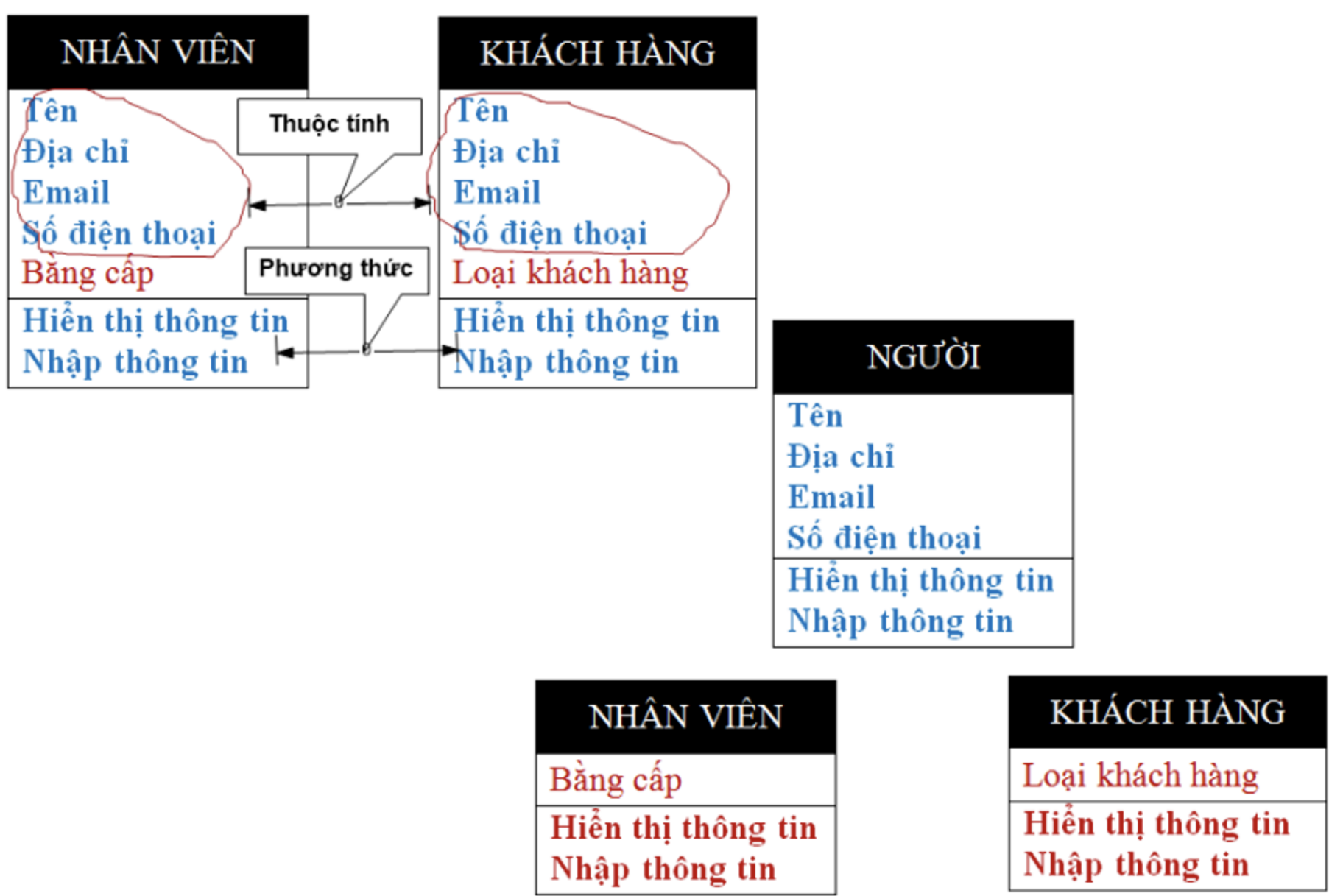


1.2. TÍNH THỪA KẾ

- ❑ Cho phép tạo ra một lớp mới kế thừa thuộc tính và phương thức của một lớp khác
- ❑ Lớp mới có thể xây dựng thêm các thuộc tính hoặc phương thức riêng
- ❑ Giúp nhất quán các thuộc tính và phương thức
- ❑ Tái sử dụng code: phần code chung định nghĩa một lần tại lớp cha, lớp con chỉ việc sử dụng mà không cần định nghĩa lại
- ❑ Thuận tiện bảo trì và phát triển



1.2. TÍNH THỪA KẾ (2)



1.2. TÍNH THỪA KẾ (3)

❑Cú pháp:

```
[<quyen_truy_cap>] class <Ten_lop_cha>
{
    ...
}

[<quyen_truy_cap>] class <Ten_lop_con> : <Ten_lop_cha>
{
    ...
}
```

- ❑Trong c# có nhiều loại thừa kế được hỗ trợ như thừa kế đơn (*Single Inheritance*), thừa kế đa cấp độ (*Single Inheritance*), thừa kế phân cấp (*Hierarchical Inheritance*)
- ❑Lớp con chỉ sử dụng được các thuộc tính và phương thức có phạm vi truy cập là public, internal, protected của lớp cha

1.2. TÍNH THỪA KẾ (4)

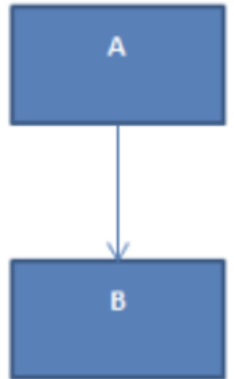
❑ **Single Inheritance:** một lớp chỉ kế thừa từ một lớp cha

```
public class NhanVien
{
    public long luong = 2000000000;
}

public class KeToan: NhanVien
{
    public long tienThuong = 5000000;
}

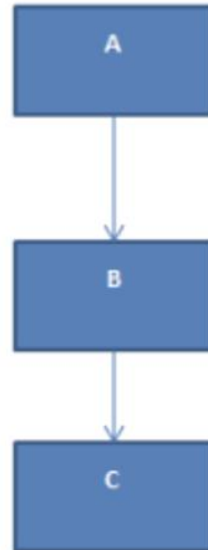
class Program
{
    public static void Main(string[] args)
    {
        KeToan kt = new KeToan();

        Console.WriteLine("Luong: " + kt.luong);
        Console.WriteLine("Tien Thuong: " + kt.tienThuong);
        Console.ReadKey();
    }
}
```



1.2. TÍNH THỪA KẾ (5)

❑ **Multilevel Inheritance:** một lớp kế thừa từ một lớp cha, mà lớp cha đó lại kế thừa từ một lớp khác



```
public class DongVat
{
    public string mauSac;
    public string tiengKeu;
}

public class Cho : DongVat
{
    public void TiengKeu()
    {
        tiengKeu = "gau gau";
        Console.WriteLine("Con cho keu: " + tiengKeu);
    }
}
```

```
public class ChoCon : Cho
{
    public void TiengKeuChoCon()
    {
        TiengKeu();
        tiengKeu = "e e";
        Console.WriteLine("Con cho con keu: " + tiengKeu);
    }
}

class Program
{
    public static void Main(string[] args)
    {
        ChoCon cc = new ChoCon();
        cc.TiengKeuChoCon();
        Console.ReadKey();
    }
}
```


1.2. TÍNH THỪA KẾ (6)

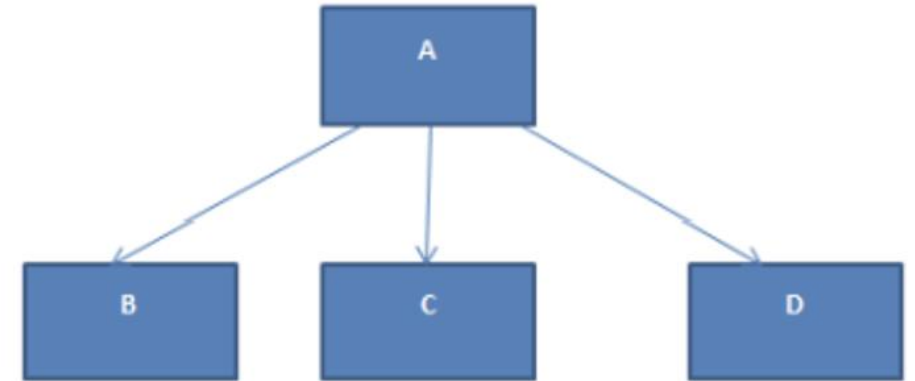
❑ **Hierarchical Inheritance:** một lớp có nhiều lớp con hay nói cách khác là có nhiều lớp cùng thừa kế từ một lớp cha

```
public class NhanVien
{
    public long luong = 20000000000;
}
public class KeToan : NhanVien
{
    public long hoaHong = 200000;
}
public class LapTrinhVien : NhanVien
{
    public long hoaHong = 500000;
}
public class QuanTriVien : NhanVien
{
    public long hoaHong = 800000;
}
```

```
class Program
{
    public static void Main(string[] args)
    {
        KeToan kt = new KeToan();
        Console.WriteLine("Luong ke toan: " + kt.luong);
        Console.WriteLine("Hoa hong ke toan: " + kt.hoaHong);

        LapTrinhVien ltv = new LapTrinhVien();
        Console.WriteLine("\nLuong lap trinh vien: " + ltv.luong);
        Console.WriteLine("Hoa hong lap trinh vien: " + ltv.hoaHong);

        QuanTriVien qtv = new QuanTriVien();
        Console.WriteLine("\nLuong quan tri Vien: " + qtv.luong);
        Console.WriteLine("Hoa hong quan tri vien: " + qtv.hoaHong);
        Console.ReadKey();
    }
}
```



1.2. TÍNH THỪA KẾ (7)

❑ Đối tượng lớp cha có thể tham chiếu đến đối tượng lớp con nhưng ngược lại thì không

```
// Tạo đối tượng con cún  
ConCun cun1 = new ConCun();    // OK  
  
DongVat cun2 = new ConCun();    // OK  
  
//ConCun cun3 = new DongVat(); // ERROR: Cannot implicitly convert type 'DongVat' to 'C'
```

❑ Dùng từ khóa base truy cập đến các thành phần bên trong lớp cha từ lớp con

❑ Không dùng từ khóa base cho các thành phần static

1.2. TÍNH THỪA KẾ (8)

- ❑ Từ khóa new trong kế thừa để xác định phạm vi của phương thức (nó không phải là toán tử khởi tạo đối tượng)
- ❑ Dùng từ khóa new cho **phương thức lớp con trùng tên phương thức lớp cha**

```
class DongVat
{
    public void Chay()
    {
        Console.WriteLine( "Dong vat chay." );
    }

    // Phương thức tĩnh
    public static void Ngủ()
    {
        Console.WriteLine( "Dong vat ngủ." );
    }
}

class ConCun : DongVat    // Kế thừa
{
    // Phương thức CÙNG TÊN với lớp cha
    public new void Chay()
    {
        // Gọi hàm từ lớp cha
        base.Chay();

        Console.WriteLine( "Con cun chay." );
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        // Tạo đối tượng Con Cún
        ConCun cun = new ConCun();

        // Gọi phương thức Chạy
        cun.Chay();

        // Gọi phương thức tĩnh bằng tên lớp
        DongVat.Ngủ();

        Console.ReadKey();
    }
}
```

1.3. TỪ KHÓA BASE

Từ khóa **base** được sử dụng để truy cập các trường, hàm khởi tạo và phương thức của lớp cơ sở. Chỉ có thể sử dụng từ khóa **base** trong phương thức, hàm khởi tạo hoặc hàm truy cập thuộc tính của đối tượng. **Không thể sử dụng nó trong phương thức tĩnh.**

Có thể sử dụng từ khóa **base** để truy cập vào các trường của lớp cơ sở trong lớp dẫn xuất. Nó rất hữu ích nếu các lớp cơ sở và các lớp dẫn xuất có cùng các trường. Nếu lớp dẫn xuất không định nghĩa cùng một trường, thì không cần sử dụng từ khóa cơ sở. Trường lớp cơ sở có thể được truy cập trực tiếp bởi lớp dẫn xuất.

Cú pháp:

```
class <ClassName>
{
    <access modifier><return type><BaseMethod>{}
}
class <ClassName1>:<ClassName>
{
    base.<BaseMethod>;
}
```

```
public class Animal
{
    public string color = "white";
}
2 references
public class Dog : Animal
{
    string color = "black";
    1 reference
    public void showColor()
    {
        Console.WriteLine(base.color);
        Console.WriteLine(color);
    }
}
0 references
public class TestBase
{
    0 references
    public static void Main()
    {
        Dog d = new Dog();
        d.showColor();
    }
}
```

1.4. TỪ KHÓA NEW

- Toán tử new để khởi tạo đối tượng
- Khai báo new modifier sẽ ẩn đi thành phần (method, property, subclass, var) được kế thừa từ base class
- new constraint áp đặt điều kiện 1 generic class bắt buộc phải có hàm tạo không tham số

```
class Person
{
    public Person(){}
}

class Program
{
    static void Main()
    {
        var person = new Person();
    }
}
```

```
public class Person
{
    public static int x = 123;
}

public class Teacher : Person
{
    new public static int x = 456;
}
```

```
class Factory where T : new() //new constraint
{
    public T Create()
    {
        return new T(); //new operator
    }
}
```

Cú pháp:

```
<access modifier> class <ClassName>
{
<access modifier> <return type><Base Method>{}
}
<access modifier> class <ClassName1>: <ClassName>
{
new <access modifier> void <Base Method>{}
}
```

Ví dụ:

```
Employees objEmp = new Employees();
```

1.5. PROTECTED ACCESS MODIFIER

```
class Employees
{
    int _empId = 1;
    string _empName = "James Anderson";
    int _age = 25;
    1 reference
    public void Display()
    {
        Console.WriteLine("Employee ID: " + _empId);
        Console.WriteLine("Employee Name: " + _empName);
    }
}

2 references
class Department : Employees
{
    int _deptID = 501;
    string _deptName = "Sales";
    1 reference
    new void Display()
    {
        base.Display();
        Console.WriteLine("Department ID: " + _deptID);
        Console.WriteLine("Department Name: " + _deptName);
    }
    0 references
    static void Main(string[] args)
    {
        Department objDepartment = new Department();
        objDepartment.Display();
    }
}
```


1.6. CONSTRUCTOR VÀ TÍNH KẾ THỪA

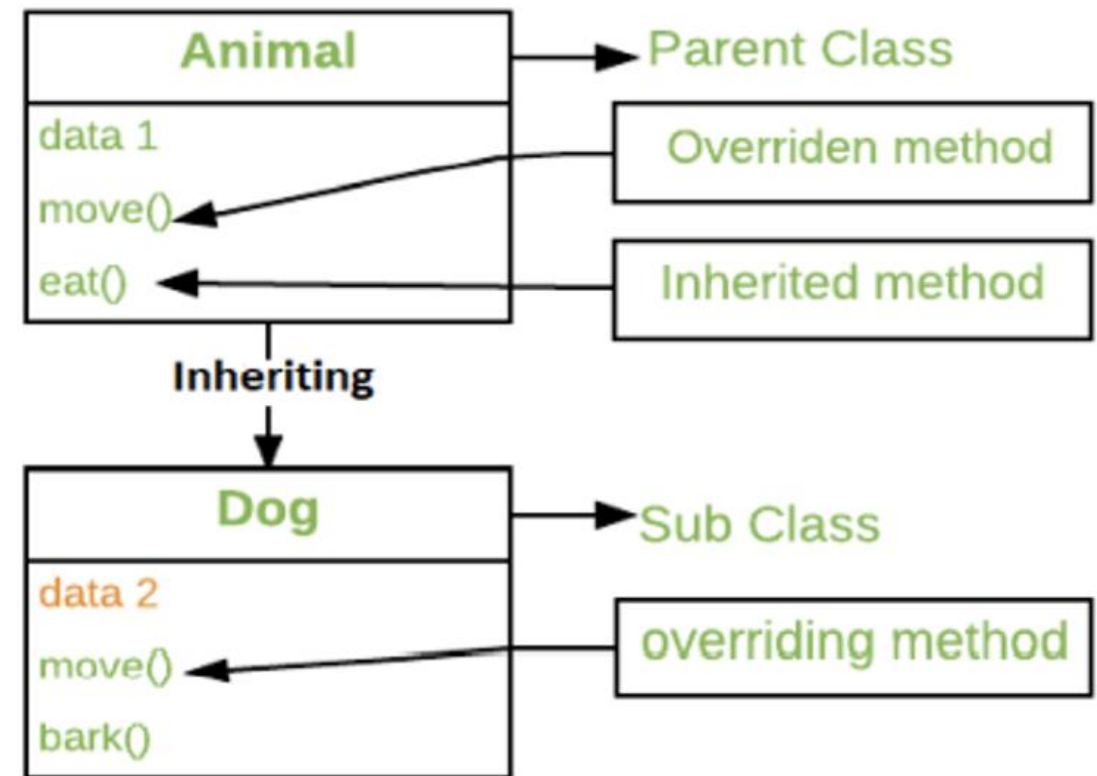
❑ Lớp cha (lớp cơ sở) có định nghĩa phương thức khởi tạo (constructor) có tham số thì tại các lớp dẫn xuất (lớp con hay lớp được kế thừa) **phải gọi phương thức khởi tạo của lớp cha**

```
class Nguoi
{
    // Khai báo thuộc tính
    public string maso;
    public string hoten;
    public string gioitinh;
    // Phương thức khởi tạo có 4 tham số
    public Nguoi(string maso, string hoten, string gioitinh)
    {
        this.maso = maso;
        this.hoten = hoten;
        this.gioitinh = gioitinh;
    }
}
```

```
class NhanVien : Nguoi
{
    private string bangcap;
    0 references
    public NhanVien(string maso, string hoten,
        string gioitinh, string bangcap)
        : base(maso, hoten, gioitinh)
    {
        this.bangcap = bangcap;
    }
}
```

2.1. OVERRIDING

- ❑ Cho phép lớp con định nghĩa lại phương thức của lớp cha
- ❑ Phương thức được **override** (ở lớp cha) và phương thức **override** (ở lớp con) phải giống hệt nhau ở cả 3 phần: kiểu dữ liệu trả về, tên phương thức và danh sách tham số
- ❑ Không thể **override** constructor.



```
class baseClass
{
    3 references
    public virtual void show()
    {
        Console.WriteLine("Base class");
    }
}
1 reference
class derived : baseClass
{
    3 references
    public override void show()
    {
        Console.WriteLine("Derived class");
    }
}
```

```
static void Main(string[] args)
{
    baseClass obj;
    obj = new baseClass();
    obj.show();
    obj = new derived();
    obj.show();
}
```

2.2. SO SÁNH OVERLOADING VÀ OVERRIDING

Override	Overload
<ul style="list-style-type: none"> – Kiểu dữ liệu trả về, tên phương thức, danh sách tham số của phương thức <i>override</i> và phương thức được <i>override</i> phải giống nhau. 	<ul style="list-style-type: none"> – Kiểu dữ liệu trả về của các phương thức <i>overload</i> có thể giống nhau hoặc khác nhau. – Số lượng tham số hoặc kiểu dữ liệu của tham số ở các phương thức <i>overload</i> phải khác nhau.
<ul style="list-style-type: none"> – Không thể thu hẹp phạm vi truy cập(<i>access modifier</i>) của phương thức được <i>override</i>. 	<ul style="list-style-type: none"> – Có thể mở rộng hoặc thu hẹp phạm vi truy cập(<i>access modifier</i>) của phương thức được <i>overload</i>.
<ul style="list-style-type: none"> – Không thể <i>overriding constructor method</i>. 	<ul style="list-style-type: none"> – <i>Overloading</i> được <i>constructor method</i>.
<ul style="list-style-type: none"> – Chỉ thực hiện được đối với các class có quan hệ kế thừa. Do đó <i>overriding</i> thực hiện ở ngoài phạm vi của một class. 	<ul style="list-style-type: none"> – Chỉ thực hiện trong cùng phạm vi, trong nội bộ của một class.
<ul style="list-style-type: none"> – Là hình thức đa hình khi chạy(<i>runtime</i>) (tức là chỉ khi chương trình chạy, thì chúng ta mới biết phương thức được gọi từ lớp nào). 	<ul style="list-style-type: none"> – Là hình thức đa hình khi biên dịch(<i>compiler</i>) (tức là khi biên dịch mới biết đang sử dụng phương thức ở trong lớp nào).
<ul style="list-style-type: none"> – Không cho phép tạo ra những ngoại lệ khác loại hoặc không phải đối tượng thuộc lớp con của lớp có thể hiện là ngoại lệ từ phương thức được <i>override</i>. 	<ul style="list-style-type: none"> – Cho phép tạo ra những ngoại lệ hoàn toàn mới so với những ngoại lệ từ phương thức được <i>overload</i>.

2.3. TỪ KHÓA VIRTUAL VÀ OVERRIDE

- **Virtual** là từ khoá dùng để khai báo 1 phương thức ảo (phương thức ảo là phương thức có thể ghi đè được).
- **Override** là từ khoá dùng để đánh dấu phương thức ghi đè lên phương thức của lớp cha.

Lưu ý:

- Chỉ có thể ghi đè lên phương thức virtual hoặc abstract
- Tính đa hình chỉ được thể hiện khi đã ghi đè lên phương thức của lớp cha.

Cú pháp virtual:

```
<access_modifier>virtual<return_type><MethodName>      (<parameter-  
list>);
```

Cú pháp override:

```
<access_modifier>override<return_type><MethodName>      (<parameter-  
list>);
```

```
using System;
public class Animal{
    public virtual void eat(){
        Console.WriteLine("Eating...");
    }
}
public class Dog: Animal
{
    public override void eat()
    {
        Console.WriteLine("Eating bread...");
    }
}
public class TestOverriding
{
    public static void Main()
    {
        Dog d = new Dog();
        d.eat();
    }
}
```

```
class Animal
{
    public void Speak()
    {
        Console.WriteLine(" Animal is speaking. . .");
    }
}

class Cat : Animal
{
    public void Speak()
    {
        Console.WriteLine(" Cat is speaking. . .");
    }
}

class Dog : Animal
{
    public void Speak()
    {
        Console.WriteLine(" Dog is speaking. . .");
    }
}
```


2.4. GỌI PHƯƠNG THỨC LỚP CƠ SỞ(BASE CLASS)

- Phương thức overriding(ghi đè) cho phép lớp dẫn xuất định nghĩa lại các phương thức của lớp cơ sở
- Cho phép các phương thức của lớp cơ sở truy cập vào phương thức mới nhưng không cho phép phương thức của lớp cơ sở nguyên bản.
- Để thực thi phương thức lớp cơ sở cũng như phương thức lớp dẫn xuất, có thể tạo một thể hiện của lớp cơ sở.
- Nó cho phép truy cập phương thức lớp cơ sở và một thể hiện của lớp dẫn xuất, để truy cập phương thức lớp dẫn xuất.

```
3 references
class Student
{
    string _studentName = "James";
    string _address = "California";
    3 references
    public virtual void PrintDetails()
    {
        Console.WriteLine("Student Name: " + _studentName);
        Console.WriteLine("Address: " + _address);
    }
}
2 references
class Grade : Student
{
    string _class = "Four";
    float _percent = 71.25F;
    3 references
    public override void PrintDetails()
    {
        Console.WriteLine("Class: " + _class);
        Console.WriteLine("Percentage: " + _percent);
    }
    0 references
    static void Main(string[] args)
    {
        Student objStudent = new Student();
        Grade objGrade = new Grade();
        objStudent.PrintDetails();
        objGrade.PrintDetails();
    }
}
```


3.1. TỪ KHÓA SEALED – NIÊM PHONG

- ❑ Một lớp được chỉ định với từ khoá sealed là một lớp không cho phép kế thừa

```
sealed class Class_Name  
{  
    //body of the class  
}
```

- ❑ Một phương thức được chỉ định với từ khoá sealed là một phương thức không cho overriding.

```
access_modifier sealed override return_type Method_Name  
{  
    //body of the method  
}
```

3 references

`sealed class Product`

```
{  
    public int Quantity;  
    public int Cost;  
}
```

0 references


`class Goods`

```
{  
    0 references  
    static void Main(string[] args)  
    {  
        Product objProduct = new Product();  
        objProduct.Quantity = 50;  
        objProduct.Cost = 75;  
        Console.WriteLine("Quantity of the Product: " + objProduct.Quantity);  
        Console.WriteLine("Cost of the Product: " + objProduct.Cost);  
    }  
}
```

0 references

`class Pen : Product`

```
{  
      
}
```

 `class Demo2.Product`

CS0509: 'Pen': cannot derive from sealed type 'Product'

Show potential fixes (Alt+Enter or Ctrl+.)

3.2. MỤC ĐÍCH CỦA LỚP SEALED

- Hãy xem xét một lớp có tên `SystemInformation` bao gồm các phương thức quan trọng ảnh hưởng đến hoạt động của hệ điều hành.
- Bạn có thể không muốn bất kỳ bên thứ ba nào kế thừa lớp `SystemInformation` và ghi đè các phương thức của nó, do đó, gây ra các vấn đề về bảo mật và bản quyền.
- Tại đây, bạn có thể khai báo lớp `SystemInformation` được niêm phong để ngăn chặn bất kỳ thay đổi nào trong các biến và phương thức của nó.
- Các lớp được niêm phong là các lớp bị hạn chế không thể kế thừa trong đó danh sách mô tả các điều kiện trong đó một lớp có thể được đánh dấu là được niêm phong:
 - Nếu ghi đè các phương thức của một lớp có thể dẫn đến hoạt động không mong muốn của lớp.
 - Khi bạn muốn ngăn bất kỳ bên thứ ba nào sửa đổi lớp của bạn.

3.3. PHƯƠNG THỨC SEALED

- Phương thức **sealed** trong C# không thể được ghi đè. Nó phải được sử dụng với từ khóa ghi đè trong phương thức.
- Khi lớp dẫn xuất ghi đè một phương thức, biến, thuộc tính hoặc sự kiện của lớp cơ sở, thì phương thức, biến, thuộc tính hoặc sự kiện mới có thể được khai báo **sealed**.
- Việc niêm phong phương thức mới ngăn không cho phương thức bị ghi đè thêm.
- Một phương thức ghi đè có thể được niêm phong bằng cách đặt trước từ khóa override bằng từ khóa **sealed**.
- **Cú pháp:**

```
sealed override <return_type> <MethodName> ()
```

```

1 reference
class ITSystem
{
    3 references
    public virtual void Print()
    {
        Console.WriteLine("The System should be handler carefully");
    }
}

1 reference
class CompanySystem:ITSystem
{
    1 reference
    public sealed override void Print()
    {
        Console.WriteLine("The system information is confidential");
        Console.WriteLine("This information should not be overridden");
    }
}

2 references
class SealedSystem : CompanySystem
{
    3 references
    public override void Print()
    {
        Console.WriteLine("This statement won't get executed");
    }
    0 references
    static void Main(string[] args)
    {
        SealedSystem objSealed=new SealedSystem();
        objSealed.Print();
    }
}

```

Error List

Entire Solution

1 Error

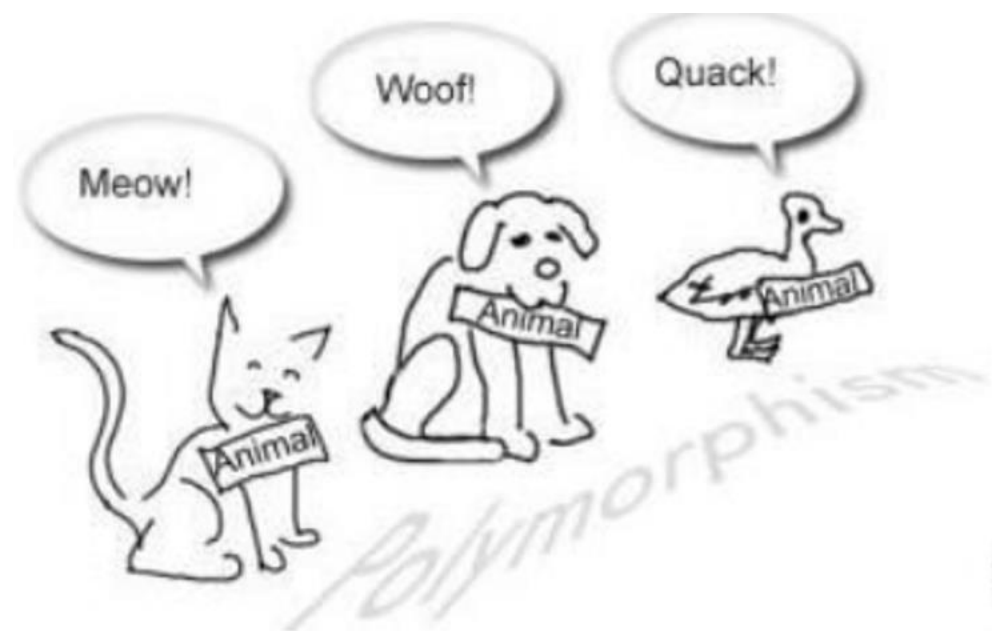
0 Warnings

0 of 21 M

	Code	Description
CS0239		'SealedSystem.Print()': cannot override inherited member 'CompanySystem.Print()' because it is sealed

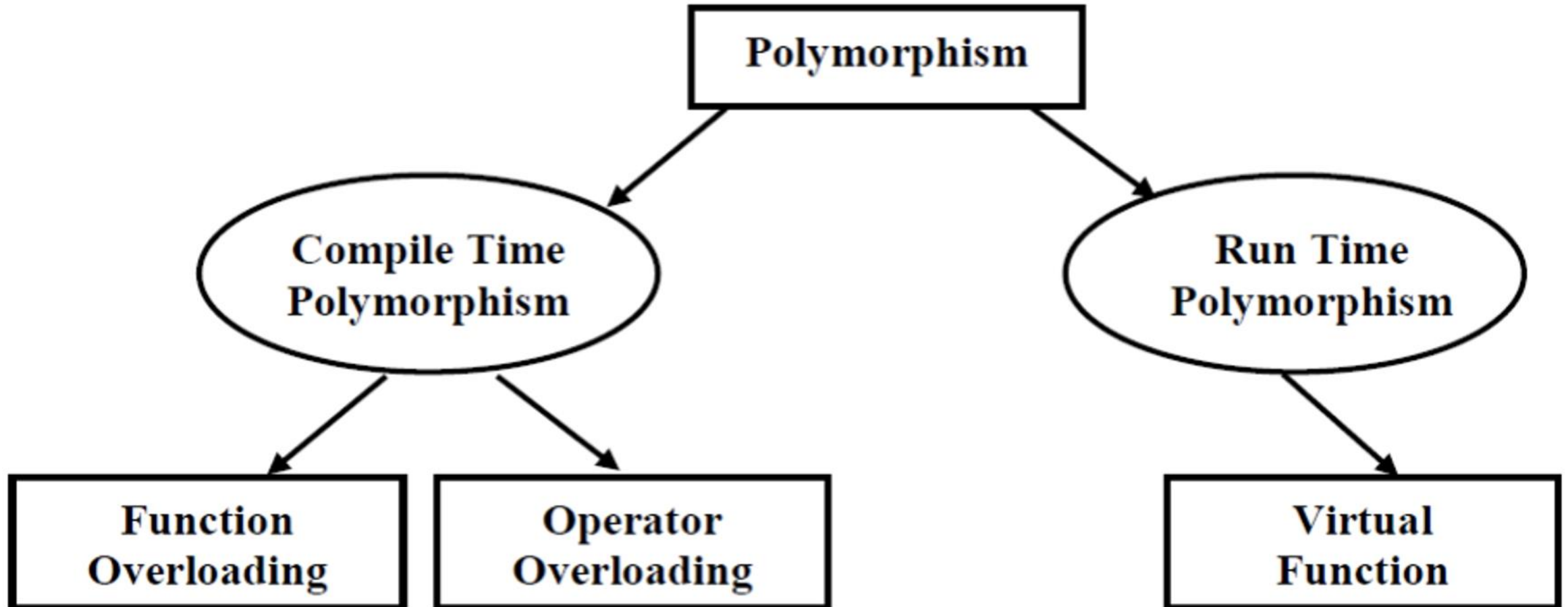
4.1. TÍNH ĐA HÌNH – POLYMORPHISM

- ❑ Tính đa hình là hiện tượng các đối tượng thuộc các lớp khác nhau có thể hiểu cùng 1 thông điệp theo các cách khác nhau
- ❑ Cho phép **một thao tác có các cách xử lý khác nhau** trên các đối tượng khác nhau



4.1. TÍNH ĐA HÌNH – POLYMORPHISM (2)

Tính đa hình được phân loại



4.2. THỰC THI

- Đa hình có thể thực thi trong C# thông qua phương thức Nạp chồng (overloading) và ghi đè (overloading).
- Bạn có thể tạo nhiều phương thức có cùng tên trong một lớp hoặc trong các lớp khác nhau có phần thân phương thức khác nhau hoặc các signatures khác nhau.
- Các phương thức có signatures giống nhau nhưng khác nhau trong một lớp được gọi là phương thức nạp chồng(overloaded). Phương thức tương tự thực hiện cùng một chức năng trên các giá trị khác nhau.
- Các phương thức được kế thừa từ lớp cơ sở trong lớp dẫn xuất và được sửa đổi trong lớp dẫn xuất được gọi là phương thức ghi đè(overridden). Chỉ phần thân của phương thức thay đổi để hoạt động theo đầu ra được yêu cầu.


```

/// <summary>
/// Method Overriding
/// </summary>
1 reference
class Hardware
{
    1 reference
    public virtual bool TurnOn()
    {
        return true;
    }
}
0 references
class Monitor: Hardware
{
    1 reference
    public override bool TurnOn()
    {
        return true;
    }
}

```

```

/// <summary>
/// Method Overloading
/// </summary>
0 references
class Hardware
{
    0 references
    public bool TurnOn()
    {
        return true;
    }
    0 references
    public bool TurnOn(string ans)
    {
        return true;
    }
}

```

0 references

class Area

{

1 reference

static int CalculateAre(int len, int wide)

{

return len * wide;

}

1 reference

static double CalculateAre(double valOne, double valTwo)

{

return 0.5 * valOne * valTwo;

}

0 references

static void Main(string[] args)

{

int length = 10;

int breadth = 22;

double tbase = 2.5;

double theight = 1.5;

Console.WriteLine("Area of Rectangle: " + CalculateAre(length, breadth));

Console.WriteLine("Area of Triangle" + CalculateAre(tbase, theight));

}

}

4.3. RUN-TIME POLYMORPHISM

❑ Đa hình khi chạy (Run-time Polymorphism) – Override.

- ❖ Các lớp phải có quan hệ kế thừa với cùng 1 lớp cha nào đó
- ❖ Phương thức đa hình phải được ghi đè (override) ở các lớp con

4.3. RUN-TIME POLYMORPHISM (2)

❑ Đa hình khi chạy (Run-time Polymorphism) – Override.

```
class Tau
{
    2 references
    public virtual void LayThongTin()
    {
        Console.WriteLine("Đây là chiếc Tàu.");
    }
}
0 references
class TauChien : Tau
{
    2 references
    public override void LayThongTin()
    {
        Console.WriteLine("Đây là Tàu chiến.");
    }
}
0 references
class TauDuLich : Tau
{
    2 references
    public override void LayThongTin()
    {
        Console.WriteLine("Đây là Tàu du lịch.");
    }
}
```

```
static void Main(string[] args)
{
    // Khai báo 3 đối tượng tàu
    Tau tau1 = new Tau();
    Tau tau2 = new TauChien();
    Tau tau3 = new TauDuLich();

    // Gọi phương thức lấy thông tin
    tau1.LayThongTin();
    tau2.LayThongTin();
    tau3.LayThongTin();
}
```

4.4. COMPILE-TIME POLYMORPHISM

❑ Đa hình khi biên dịch (Compile-time Polymorphism)

❖ Tính đa hình khi biên dịch thể hiện ở sự đa dạng nhờ sự khác biệt :

- Số lượng tham số
- Kiểu dữ liệu của tham số

```
public class Calculate
{
    public void AddNumbers(int a, int b)
    {
        Console.WriteLine("a + b = {0}", a + b);
    }
    public void AddNumbers(int a, int b, int c)
    {
        Console.WriteLine("a + b + c = {0}", a + b + c);
    }
}
class Program
{
    static void Main(string[] args)
    {
        Calculate c = new Calculate();
        c.AddNumbers(1, 2);
        c.AddNumbers(1, 2, 3);
        Console.WriteLine("\nPress Enter Key to Exit..");
        Console.ReadLine();
    }
}
```

4.5. SO SÁNH RUN-TIME VÀ COMPILE-TIME POLYMORPHISM

Compile-time Polymorphism	Run-time Polymorphism
Thực thi thông qua phương thức overloading	Thực thi thông qua phương thức overridden
Được thực thi tại thời điểm biên dịch vì trình biên dịch biết phương thức nào sẽ thực thi tùy thuộc vào số lượng tham số và kiểu dữ liệu của chúng.	Được thực thi tại thời điểm chạy vì trình biên dịch không biết phương thức được thực thi, cho dù đó là phương thức lớp cơ sở sẽ được gọi hay phương thức lớp dẫn xuất.
Được gọi là đa hình tĩnh(static)	Được gọi là đa hình động(dynamic)

5 references

```
class Circle
{
    protected const double PI = 3.14;
    protected const double Radius = 14.9;
    3 references
    public virtual double Area()
    {
        return PI * Radius * Radius;
    }
}
```

0 references

```
class Cone : Circle
{
    protected double Side = 10.2;
    3 references
    public override double Area()
    {
        return PI * Radius * Side;
    }
    0 references
    static void Main(string[] args)
    {
        Circle objRunOne = new Circle();
        Console.WriteLine("Area is: " + objRunOne.Area());
        Circle objRunTwo = new Circle();
        Console.WriteLine("Area is: " + objRunTwo.Area());
    }
}
```

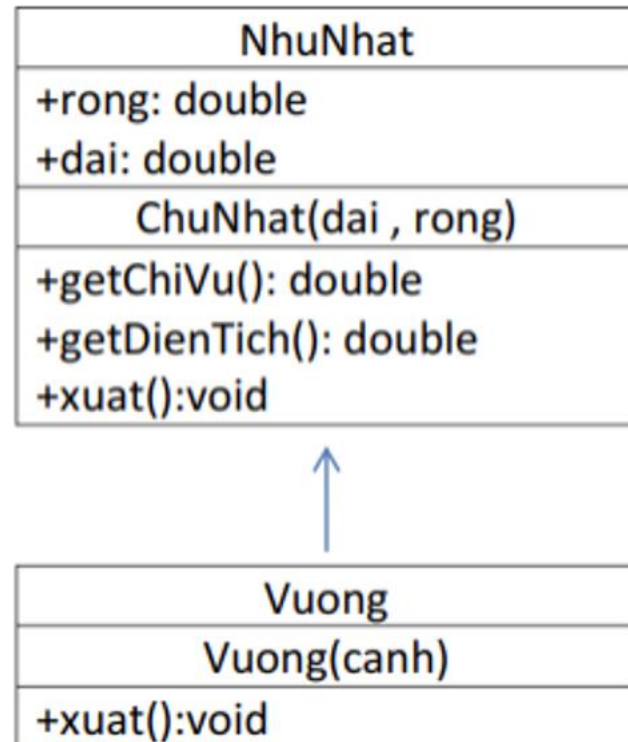
BÀI TẬP

Bài 1: Xây dựng lớp ChuNhat gồm 2 thuộc tính là rong và dai và các phương thức getChuVi() và getDienTich() để tính chu vi và diện tích. Phương thức xuat() sẽ xuất ra màn hình chiều rộng, chiều dài, diện tích và chu vi.

Xây dựng lớp Vuong kế thừa từ lớp ChuNhat và ghi đè phương thức xuat() để xuất thông tin cạnh, diện tích và chu vi. Viết chương trình nhập 2 hình chữ nhật và một hình vuông sau đó xuất ra màn hình.

HƯỚNG DẪN

- Xây dựng các lớp theo kiến trúc phân cấp kế thừa như sau



```
ChuNhat cn = new ChuNhat(dai, rong)
ChuNhat vu = new Vuong(canh)
cn.xuat();
vu.xuat()
```

- Hàm tạo của lớp **Vuong** gọi lại hàm tạo của lớp **ChuNhat** và truyền cạnh của hình vuông cho 2 tham số chiều dài và chiều rộng
- Ghi đè phương thức xuất để xuất thông tin của hình vuông
- Tạo lớp chứa phương thức `main()` và nhập chiều dài, chiều rộng của hình chữ nhật và cạnh của hình vuông. Sau đó sử dụng các lớp **ChuNhat** và **Vuong** để tạo các đối tượng và gọi phương thức `xuat()` để xem thông tin: