

# Object-Oriented Programming Using Microsoft Visual C# .NET

## Inheritance

# C# Classes

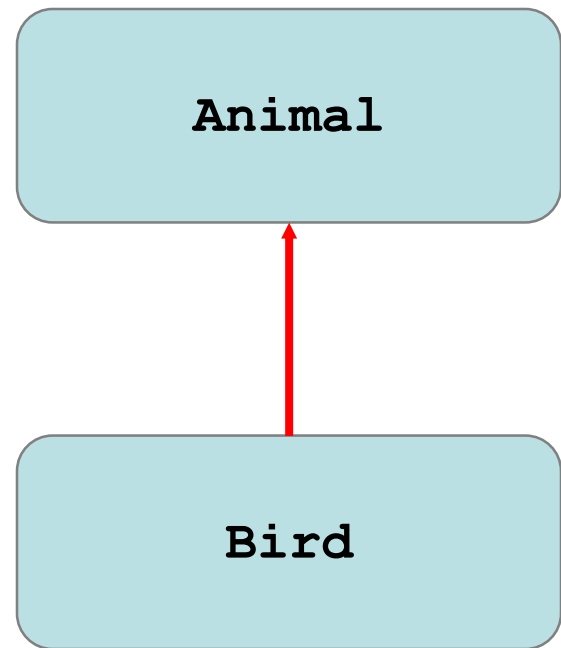
- Classes are used to accomplish:
  - Modularity: Scope for global (static) methods
  - Blueprints for generating objects or instances:
    - Per instance data and method signatures
- Classes support
  - Data encapsulation - private data and implementation.
  - Inheritance - code reuse

# Inheritance

- Inheritance allows a software developer to derive a new class from an existing one.
- The existing class is called the parent, super, or base class.
- The derived class is called a child or subclass.
- The child inherits characteristics of the parent.
  - Methods and data defined for the parent class.
- The child has special rights to the parents methods and data.
  - Public access like any one else
  - *Protected* access available only to child classes (and their descendants).
- The child has its own unique behaviors and data.

# Inheritance

- Inheritance relationships are often shown graphically in a *class diagram*, with the arrow pointing to the parent class.
- Inheritance should create an *is-a* relationship, meaning the child *is a* more specific version of the parent.



# Examples: Base Classes and Derived Classes

Base class	Derived classes
Student	GraduateStudent UndergraduateStudent
Shape	Circle Triangle Rectangle
Loan	CarLoan HomeImprovementLoan MortgageLoan
Employee	FacultyMember StaffMember
Account	CheckingAccount SavingsAccount

# Declaring a Derived Class

- Define a new class `DerivedClass` which extends `BaseClass`

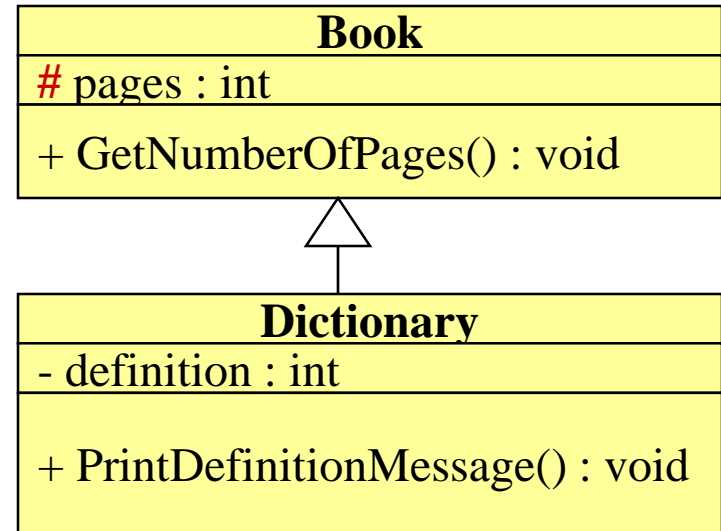
```
class BaseClass
{
    // class contents
}
class DerivedClass : BaseClass
{
    // class contents
}
```

# Controlling Inheritance

- A child class inherits the methods and data defined for the parent class; however, whether a data or method member of a parent class is accessible in the child class depends on the visibility modifier of a member.
- Variables and methods declared with *private* visibility are not accessible in the child class
  - However, a private data member defined in the parent class is still part of the state of a derived class.
- Variables and methods declared with *public* visibility are accessible; but public variables violate our goal of encapsulation
- There is a third visibility modifier that helps in inheritance situations: *protected*.

# The protected Modifier

- Variables and methods declared with protected visibility in a parent class are only accessible by a child class or any class derived from that class



+ public  
- private  
# protected



# Single Inheritance

- Some languages, e.g., C++, allow *Multiple inheritance*, which allows a class to be derived from two or more classes, inheriting the members of all parents.
- C# and Java support *single inheritance*, meaning that a derived class can have only one parent class.

# Overriding Methods

- A child class can *override* the definition of an inherited method in favor of its own
- That is, a child can redefine a method that it inherits from its parent
- The new method must have the same signature as the parent's method, but can have a different implementation.
- The type of the object executing the method determines which version of the method is invoked.

# Virtual Methods

- A method, which can be overridden, is called **virtual**.
- This means **changing their implementation**.
  - the original source code from the base class is ignored and new code takes its place.
- If we want a method to be overridable, we can do so by including the keyword **virtual** in the declaration of the method.
- One of the fundamental principle of Object-Oriented Programming is "**Polymorphism**".
  - it is mostly related to **overriding methods in derived classes**, in order **to change their original behavior** inherited from the base class.

# Virtual Methods

- If we want to make a method virtual, we mark it with the **keyword *virtual***. Then the derived class can declare and define a method with the **same signature**.
- A method marked with the keyword *override* is automatically **virtual** too. ( Its derived class can declare and define a method with the same signature.)

# Override methods

- Rules:
  - **Private** method can not be *virtual* or *override*
  - The signature of the virtual and override methods must be the same.
  - The access modifier of the virtual and the override methods must be the same.
  - To **override** a base method, the base method must be defined as **virtual**, **abstract**, or **override**.
  - When a derived class contains a method that overrides a parent class method, you might have occasion to use the parent class version of the method within the subclass. If so, you can use the keyword **base** to access the parent class method.

# To the example: Hunting dogs (Pointers-retrievers)



Hungarian Vizsla  
Hungarian Pointer  
Magyar Vizsla



Drótszőrű magyar vizsla,  
Hungarian Wirehaired Vizsla

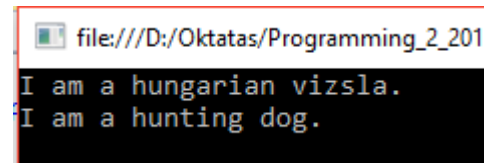
# An example

A virtual method (or property) is one that can be overridden by a method with the same signature in a child class.

```
class Dog {  
    public virtual void WhoAreYou() { Console.WriteLine("I am a dog."); }  
}  
class HuntingDog : Dog{  
    public override void WhoAreYou() { Console.WriteLine("I am a hunting dog"); }  
}  
class HungarianVizsla : HuntingDog {  
    public override void WhoAreYou() { Console.WriteLine("I am a hungarian vizsla"); }  
}
```

```
Dog dog1 = new HungarianVizsla(); // allow  
dog1.WhoAreYou();                // "I am a hungarian vizsla"
```

```
Dog dog2 = new HuntingDog();      // allow  
dog2.WhoAreYou();                // "I am a hunting dog"
```



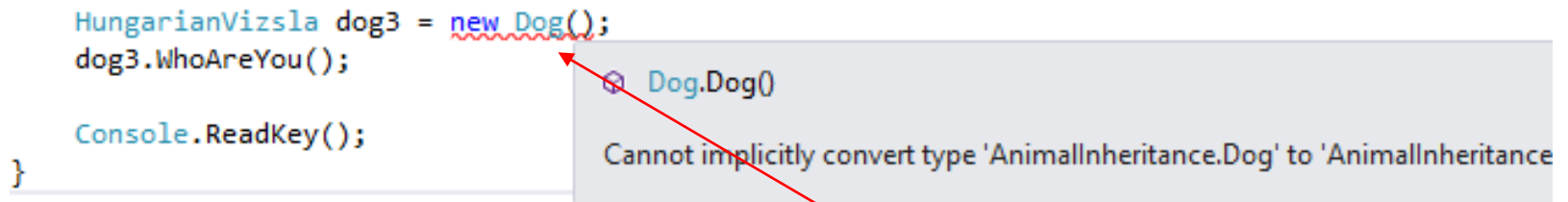
```
file:///D:/Oktatas/Programming_2_201  
I am a hungarian vizsla.  
I am a hunting dog.
```

- Every derived class object “is a” specific instance of both the derived class and the base class.
- You can assign derived class object to an object of any of its parent class types. When you do, C# makes an implicit reference conversion from derived class to base class.
- Example:

```
Dog dog1 = new HungarianVizsla();
```

After invoke the *new* operator *dog1* object behaves as a *HungarianVizsla* object (it is really a hungarianvizsla object), can call the methods and use Properties of HingarianVizsla class. It is important that this conversion is not working in the opposite direction as you can see:

```
HungarianVizsla dog3 = new Dog();  
dog3.WhoAreYou();  
  
Console.ReadKey();  
}
```



The editor notes!



# Typecasting

Declare a Dog type **dog1** name variable and invoke the HungarianVizsla() constructor to create an object. If we want to call the Hunt() method of the HungarianVizsla class we cannot, because only during compilation will be clear that the Dog type **dog1** is really a HungarianVizsla object. In this case we must use explicit type conversion.

```
Dog dog1 = new HungarianVizsla();
```

```
dog1.|
```

- Equals
- GetHashCode
- GetType
- ToString
- WhoAreYou `void Dog.WhoAreYou()`

```
Dog dog1 = new HungarianVizsla();
```

```
((HungarianVizsla)dog1).|
```

- Equals
- GetHashCode
- GetType
- Hunt
- ToString
- WhoAreYou

```
Dog dog1 = new HungarianVizsla();  
//Explicit type conversion  
((HungarianVizsla)dog1).Hunt();
```

# Typecasting

- You can use the **is** operator to check whether a conversion would complete successfully.

```
Dog dog1 = new HungarianVizsla();  
//Explicit type conversion  
if (dog1 is HungarianVizsla) {  
    ((HungarianVizsla)dog1).Hunt();  
}  
dog1.WhoAreYou();
```

The **as** operator is like the cast operator, except that it does not raise an exception. If the conversion fails, rather than raising an exception, it returns null.

```
if (dog1 is HungarianVizsla) {  
    (dog1 as HungarianVizsla).Hunt();  
}
```

[3 references](#)

```
class Dog {  
    4 references  
    public virtual void WhoAreYou() {  
        Console.WriteLine("I am a dog.");  
    }  
}
```

[2 references](#)

```
class HuntingDog : Dog {  
    4 references  
    public override void WhoAreYou()  
    {  
        Console.WriteLine("I am a hunting dog.");  
    }  
}
```

[1 reference](#)

```
class HungarianVizsla : HuntingDog {  
    4 references  
    public override void WhoAreYou()  
    {  
        Console.WriteLine("I am a hungarian vizsla.");  
    }  
    0 references  
    public void Hunt() {  
        Console.WriteLine("I am a very good hunter.");  
    }  
}
```

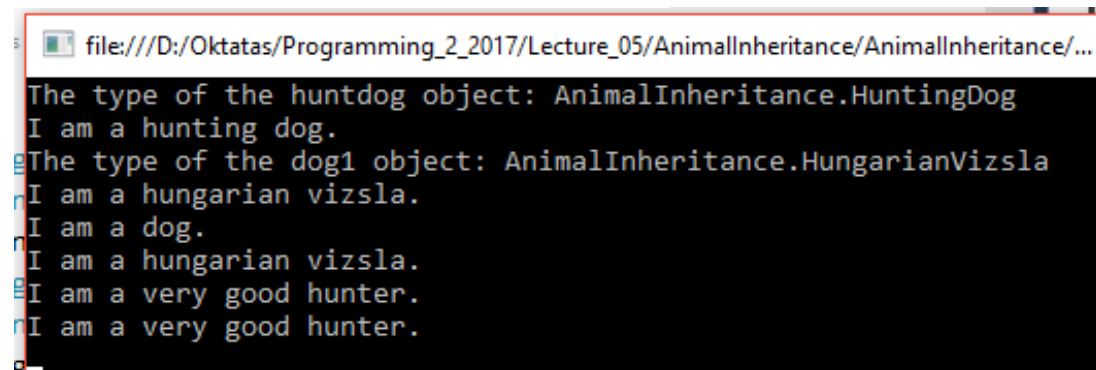
## The classes

```
file:///D:/Oktatas/Programming_2_2017/Lectu  
I am a hungarian vizsla.  
I am a very good hunter.  
I am a hunting dog.  
_
```

```
static void Main(string[] args)
{
    Dog huntDog = new HuntingDog();
    Console.WriteLine("The type of the huntDog object: " + huntDog.GetType());
    huntDog.WhoAreYou();
    Dog dog1 = new HungarianVizsla();
    Console.WriteLine("The type of the dog1 object: " + dog1.GetType());
    dog1.WhoAreYou();
    Dog dog = new Dog();
    dog.WhoAreYou();

    Dog vizs = new HungarianVizsla();
    vizs.WhoAreYou();
    //Explicit type conversion
    if (vizs is HungarianVizsla) {
        ((HungarianVizsla)vizs).Hunt();
    }

    // other mode of type conversion
    if (vizs is HungarianVizsla) {
        (vizs as HungarianVizsla).Hunt();
    }
    Console.ReadKey();
}
```



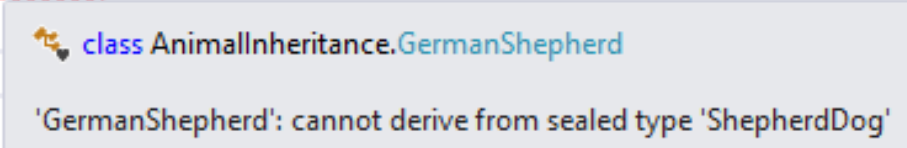
```
file:///D:/Oktatas/Programming_2_2017/Lecture_05/AnimalInheritance/AnimalInheritance/...
The type of the huntDog object: AnimalInheritance.HuntingDog
I am a hunting dog.
The type of the dog1 object: AnimalInheritance.HungarianVizsla
I am a hungarian vizsla.
I am a dog.
I am a hungarian vizsla.
I am a very good hunter.
I am a very good hunter.
```

# Sealed classes and methods

- A sealed class can be instantiated only as a stand-alone class object—it cannot be used as a base class. Sealed class is labeled with the sealed modifier:

```
1 reference
sealed class ShepherdDog : Dog {

}
0 references
class GermanShepherd : ShepherdDog {
}
0 references
```



# Sealed class and methods

- **Sealing of methods** is done when we rely on a piece of functionality and we don't want it to be altered. We already know that methods are **sealed** by default. But if we want a base class' virtual method to become sealed in a derived class, we use **sealed override**.

```
class Dog : Animal
{
    public sealed override void Eat()
    {
        Console.WriteLine("Vau - Vau - Hamm - Hamm");
    }
}

sealed class Dobermann : Dog
{
    public override void Eat() // ez sem jó
    {
    }
}
```

# Sealed class

If the class we want to inherit is marked with the keyword *sealed*, inheritance is not possible.

The type **string** is sealed, so it cannot be inherited.

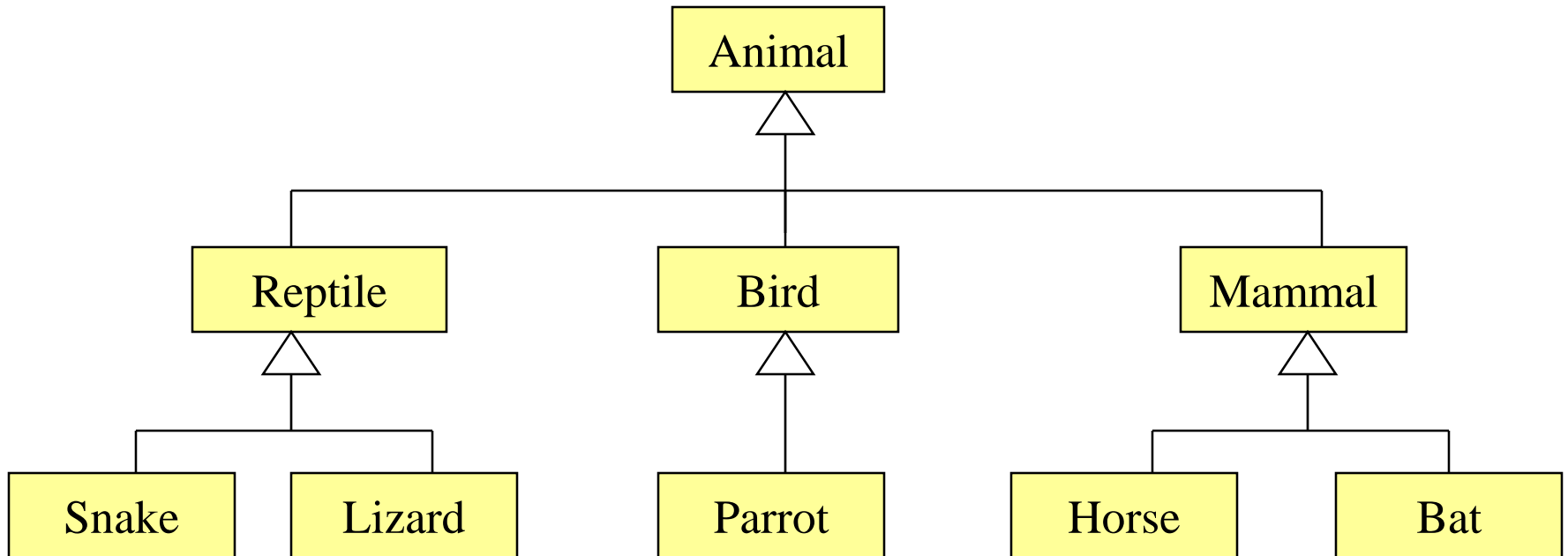
# Sealed methods

- The relationship of virtual, override, sealed keywords :
  - A **virtual** method is the *first implementation* of the method.
  - An **override** method is the *other implementation* of the base method.
  - A **sealed** method is the *last implementation* of the base method.

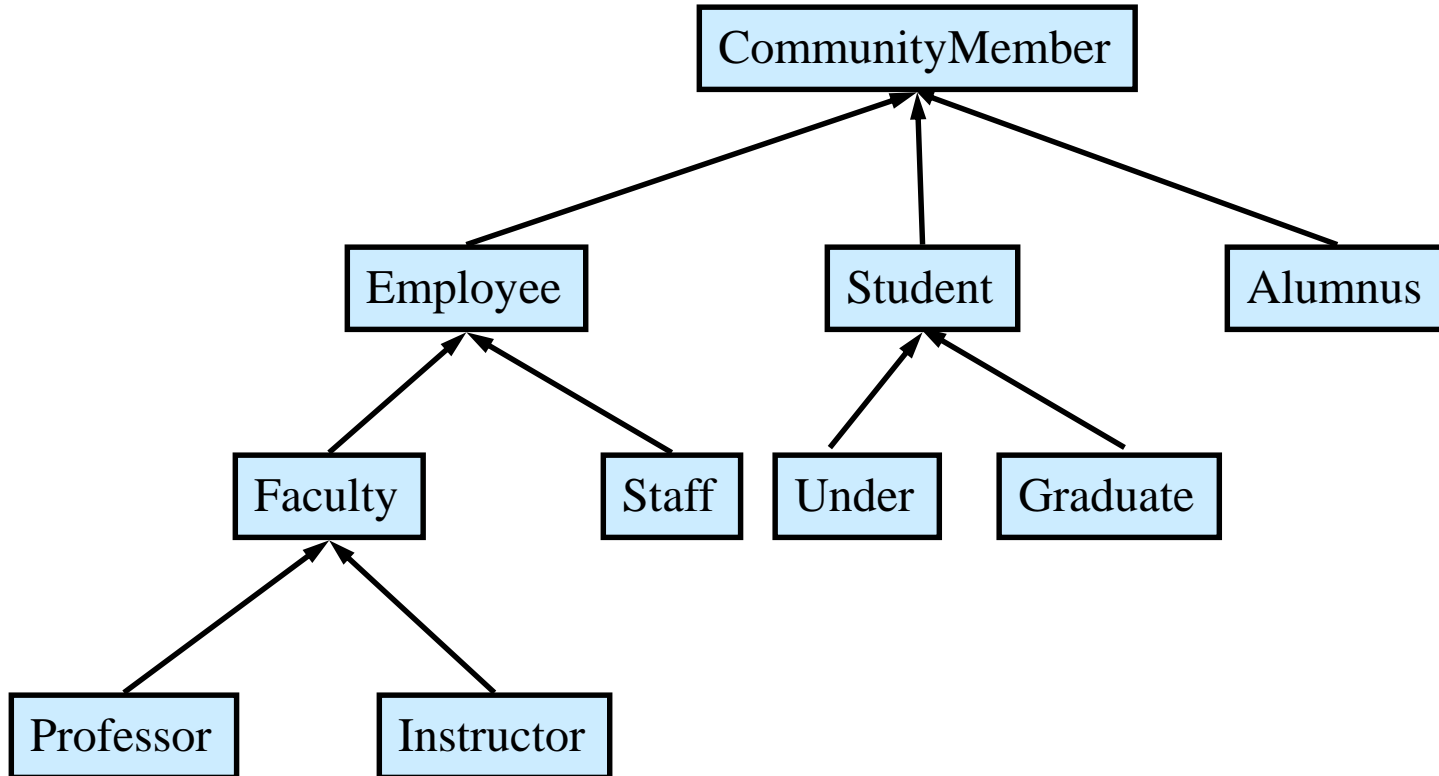


# Class Hierarchies

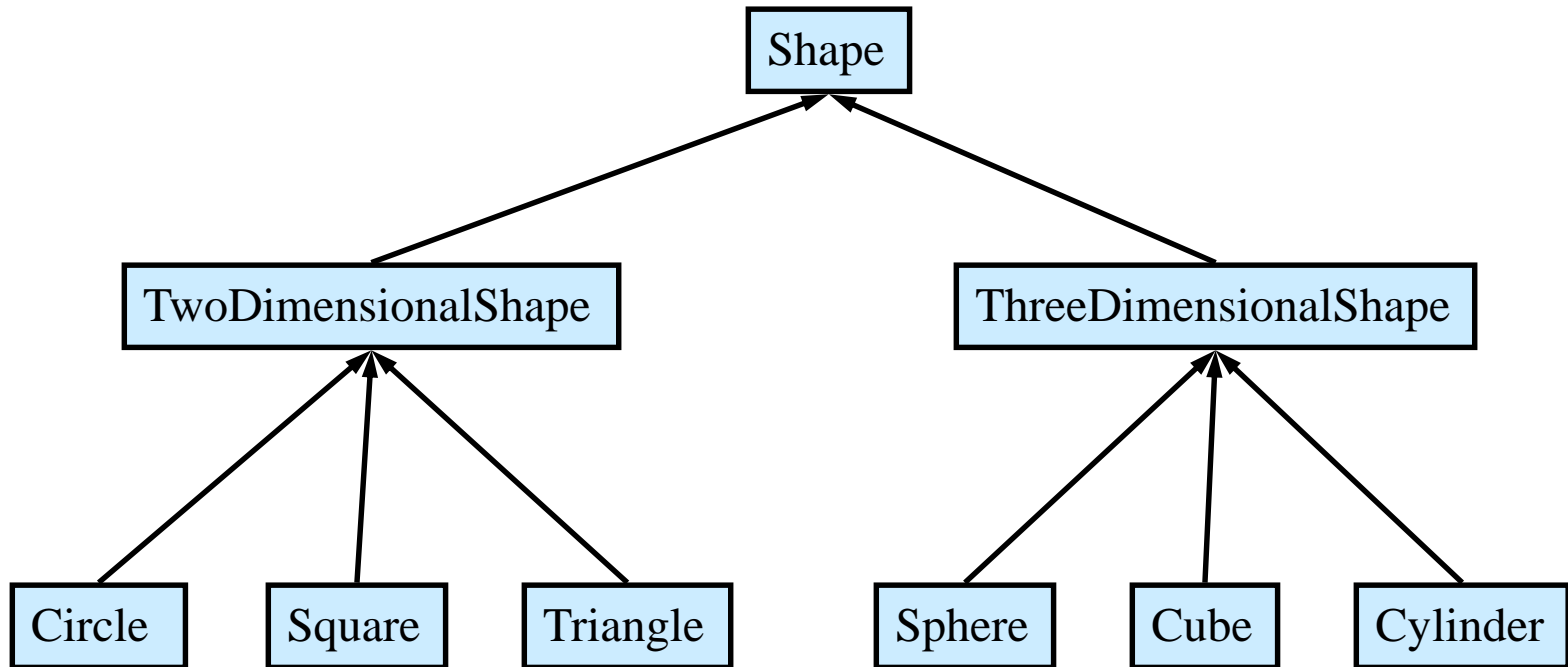
- A child class of one parent can be the parent of another child, forming a *class hierarchy*



# Class Hierarchies



# Class Hierarchies



# Class Hierarchies

- An inherited member is continually passed down the line
  - Inheritance is transitive.
- Good class design puts all common features as high in the hierarchy as is reasonable. Avoids redundant code.

# References and Inheritance

- An object reference can refer to an object of its class, or to an object of any class derived from it by inheritance.
- For example, if the `Holiday` class is used to derive a child class called `Christmas`, then a `Holiday` reference can be used to point to a `Christmas` object.

```
Holiday day;  
day = new Holiday();  
...  
day = new Christmas();
```

# Dynamic Binding

- A polymorphic reference is one which can refer to different types of objects at different times. It morphs!
- The type of the actual instance, not the declared type, determines which method is invoked.
- Polymorphic references are therefore resolved at *run-time*, not during compilation.
  - This is called ***dynamic binding***.

# Dynamic Binding

- Suppose the `Holiday` class has a method called `Celebrate`, and the `Christmas` class redefines it (overrides it).
- Now consider the following invocation:  

```
day.Celebrate();
```
- If `day` refers to a `Holiday` object, it invokes the `Holiday` version of `Celebrate`; if it refers to a `Christmas` object, it invokes the `Christmas` version

# Overriding Methods

- C# requires that all class definitions communicate clearly their intentions.
- The keywords *virtual*, *override* and *new* provide this communication.
- If a base class method is going to be overridden it should be declared *virtual*.
- A derived class would then indicate that it indeed does override the method with the *override* keyword.



# Overriding Methods

- If a derived class wishes to hide a method in the parent class, it will use the *new* keyword.
- This should be avoided.

# Overloading vs. Overriding

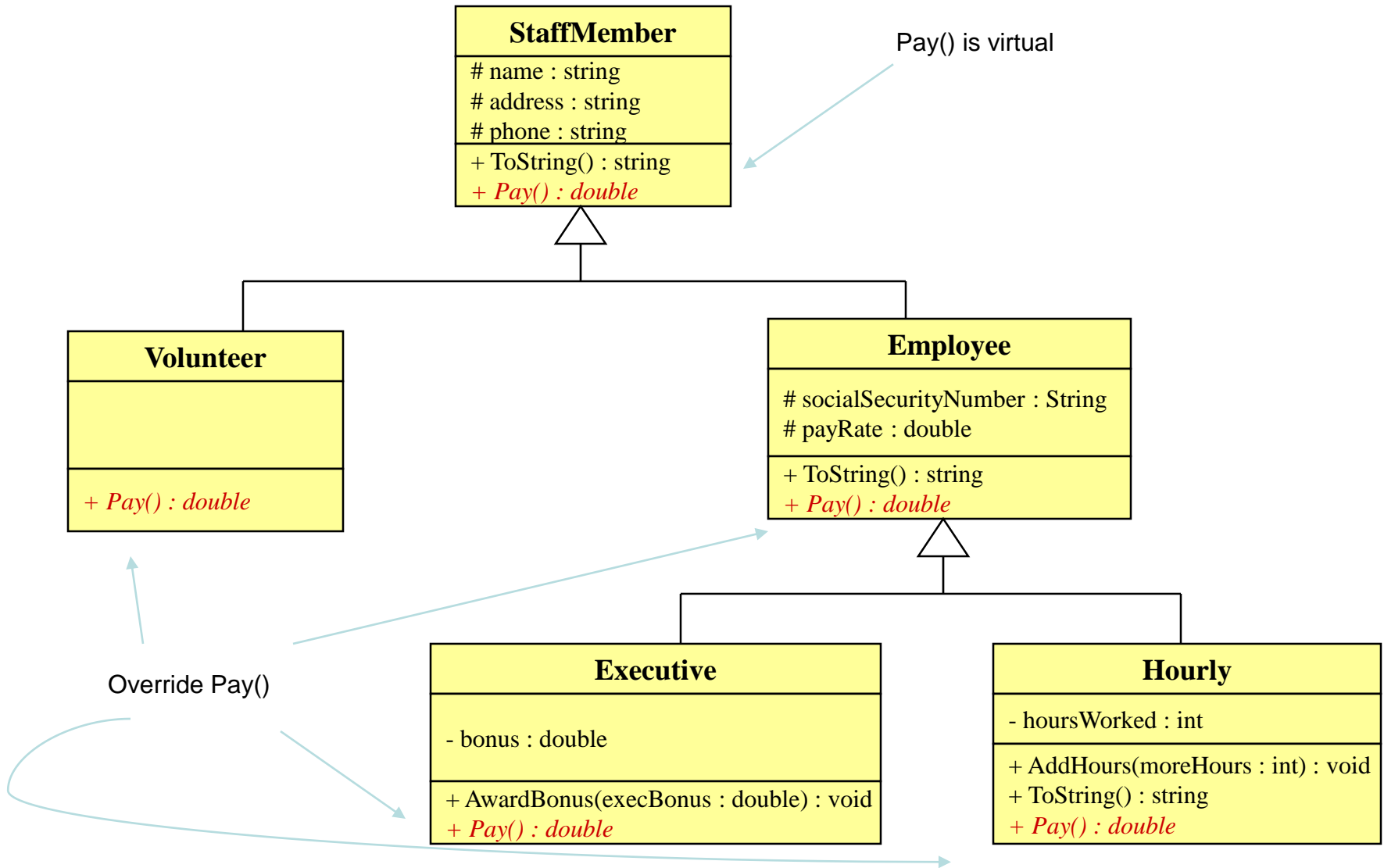
- **Overloading** deals with multiple methods in the same class with the same name but different signatures
- **Overloading** lets you define a similar operation in different ways for different data
- Example:  

```
int foo(string[] bar);  
int foo(int bar1, float a);
```

- **Overriding** deals with two methods, one in a parent class and one in a child class, that have the same signature
- **Overriding** lets you define a similar operation in different ways for different object types
- Example:

```
class Base {  
    public virtual int foo() {}  
}  
class Derived {  
    public override int foo() {}  
}
```

# Polymorphism via Inheritance



# Widening and Narrowing

- Assigning an object to an ancestor reference is considered to be a **widening** conversion, and can be performed by simple assignment

```
Holiday day = new Christmas();
```

- Assigning an ancestor object to a reference can also be done, but it is considered to be a **narrowing** conversion and must be done with a cast:

```
Christmas christ = new Christmas();
```

```
Holiday day = christ;
```

```
Christmas christ2 = (Christmas)day;
```

# Widening and Narrowing

- Widening conversions are most common.
  - Used in polymorphism.
- Note: Do not be confused with the term widening or narrowing and memory. Many books use *short* to *long* as a widening conversion. A *long* just happens to take-up more memory in this case.
- More accurately, think in terms of sets:
  - The set of animals is greater than the set of parrots.
  - The set of whole numbers between 0-65535 (*ushort*) is greater (wider) than those from 0-255 (*byte*).

# The `System.Object` Class

- All classes in C# are derived from the `Object` class
  - if a class is not explicitly defined to be the child of an existing class, it is a direct descendant of the `Object` class
- The `Object` class is therefore the ultimate root of all class hierarchies.
- The `Object` class defines methods that will be shared by all objects in C#, e.g.,
  - `ToString`: converts an object to a string representation
  - `Equals`: checks if two objects are the same
  - `GetType`: returns the type of a type of object
- A class can override a method defined in `Object` to have a different behavior, e.g.,
  - `String` class overrides the `Equals` method to compare the content of two strings

# Abstract Classes and methods

- Abstract classes
  - Cannot be instantiated
  - Used as base classes
  - Class definitions are **not complete**
    - Derived classes must define the missing pieces
  - Can contain **abstract methods** and/or **abstract properties**
    - Have **no implementation**
    - Derived classes **must override inherited abstract methods** and **abstract properties** to enable instantiation
      - Abstract methods and abstract properties are **implicitly virtual**

# Abstract classes and methods

- An abstract class is used to provide an appropriate base class from which other classes may inherit (concrete classes)
- Abstract base classes are too generic to define (by instantiation) real objects
- To define an abstract class, use keyword **abstract** in the declaration
- To declare a method or property abstract, use keyword **abstract** in the declaration
- Abstract methods and properties have no implementation



# Abstract classes and methods

- Concrete classes use the keyword **override** to provide implementations for all the abstract methods and properties of the base-class
- Any class with an abstract method or property must be declared **abstract**
- Even though abstract classes cannot be instantiated, we can use abstract class references to refer to instances of any concrete class derived from the abstract class

# DATA FORMATTING

# Formatting output in C#

- Format specifiers can be use in the **Console** class output methods (***Write()***, ***WriteLine()***), and in the **String** class ***Format()*** method and in the ***ToString()*** method.
- The format specifier is placed inside the curly braces ({ }), thus becoming part of the **string** literal argument for the **WriteLine()** method.
- Notice two values are placed inside the braces. The first value in the curly brace is a placeholder. It indicates which of the arguments that are placed outside of the double quotes you want displayed:

```
int n = 122334;
```

```
Console.WriteLine("{0:D} = {1:X}", n, n);
```

```
Console.WriteLine(String.Format("{0:D} = {1:X}", n, n));
```

```
Console.WriteLine("{0} = {1}",n.ToString("D"), n.ToString("X"));
```

# Standard numeric format specifiers

Character	Name	Format specifier
C, c	Currency	{0:c}
D, d	Decimal	{0:d}
E, e	Scientific (exponent)	{0:e}
F, f	Fixed point	{0:f}
G, g	General	{0:g}
N, n	Number with thousand separator	{0:n}
R, r	Rounded	{0:r}
P, p	Percent	{0:p}
X, x	Hexadecimal	{0:X}

# Precision specifier

Format character	Name	Format specifier with precision
C, c	Currency	{0:c7}
D, d	Decimal	{0:d7}
E, e	Scientific	{0:e7}
F, f	Fix-point	{0:f7}
G, g	General	{0:g7}
N, n	Number with thousand separator	{0:n7}
R, r	Round-trip	{0:r7}
P, p	Percent	{0:p7}
X, x	Hexadecimal	{0:X7}

**Precision:** the number of significant digits or zeros to the right of the decimal point.

You may also specify a width as part of the format specifier. This is especially useful when you want to control the alignment of items on multiple lines. Add the Alignment component following the index ordinal before the colon. A comma is used as a separator. If the value of alignment is less than the length of the formatted string, alignment is ignored and the length of the formatted string is used as the field width. The formatted data in the field is right-aligned if alignment is positive and left-aligned if alignment is negative. If padding is necessary, white space is used.

Format character	Name	Format specifier
C, c	Currency	{0,12:c7}
D, d	Decimal	{0,12:d7}
E, e	Scientific	{0,12:e7}
F, f	Fix-point	{0,12:f7}
G, g	General	{0,12:g7}
N, n	Number with thousand separator	{0,12:n7}
R, r	Round-trip	{0,12:r3}
P, p	Percent	{0,12:p7}
X, x	Hexadecimal	{0,12:X7}

```
class Program
```

```
{
```

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

```
        int n = 124239,m=34;
```

```
        double y = 28.783457687;
```

```
        Console.WriteLine("n={0:c}\t\t n={1:c3}",n, n);
```

```
        Console.WriteLine("n={0:d}\t\t n={1:d3}",n, n);
```

```
        Console.WriteLine("n={0:e}\t\t n={1:e3}",n, n);
```

```
        Console.WriteLine("n={0:f}\t\t n={1:f3}",n, n);
```

```
        Console.WriteLine("n={0:g}\t\t n={1:g3}",n, n);
```

```
        Console.WriteLine("n={0:n}\t\t n={1:n3}",n, n);
```

```
        Console.WriteLine("m={0:p}\t\t m={1:p3}",m, m);
```

```
        Console.WriteLine("n={0:x}\t\t n={1:x8}",n, n);
```

```
n=124 239,00 Ft          n=124 239,000 Ft
n=124239                n=124239
n=1,242390e+005         n=1,242e+005
n=124239,00            n=124239,000
n=124239                n=1,24e+05
n=124 239,00           n=124 239,000
m=3 400,00 %           m=3 400,000 %
n=1e54f                 n=0001e54f
```

```

// precision (number of significant digits) and with
Console.WriteLine("\nwith and precision");
Console.WriteLine("n={0,10:c3}", n);
Console.WriteLine("n={0,10:d3}", n);
Console.WriteLine("n={0,-10:d3} left align", n);
Console.WriteLine("n={0,10:e3}", n);
Console.WriteLine("n={0,10:f3}", n);
Console.WriteLine("n={0,10:g3}", n);
Console.WriteLine("n={0,10:n3}", n);
Console.WriteLine("n={0,10:p3}", n);
Console.WriteLine("n={0,10:x4}", n);

```

```

with and precision
n=124 239,000 Ft
n=      124239
n=124239 left align
n=1,242e+005
n=124239,000
n=  1,24e+05
n=124 239,000
n=12 423 900,000%
n=      1e54f

```



# Continue sample

```
// real numbers  
Console.WriteLine("real number without formatting:\t{0}", y);  
Console.WriteLine("real number with formatting:\t{0,10:f7}", y);  
Console.ReadKey();
```



```
real number without formatting: 28,783457687  
real number with formatting:   28,7834577
```

Thank you for your attention!