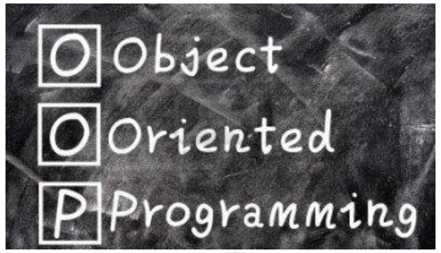


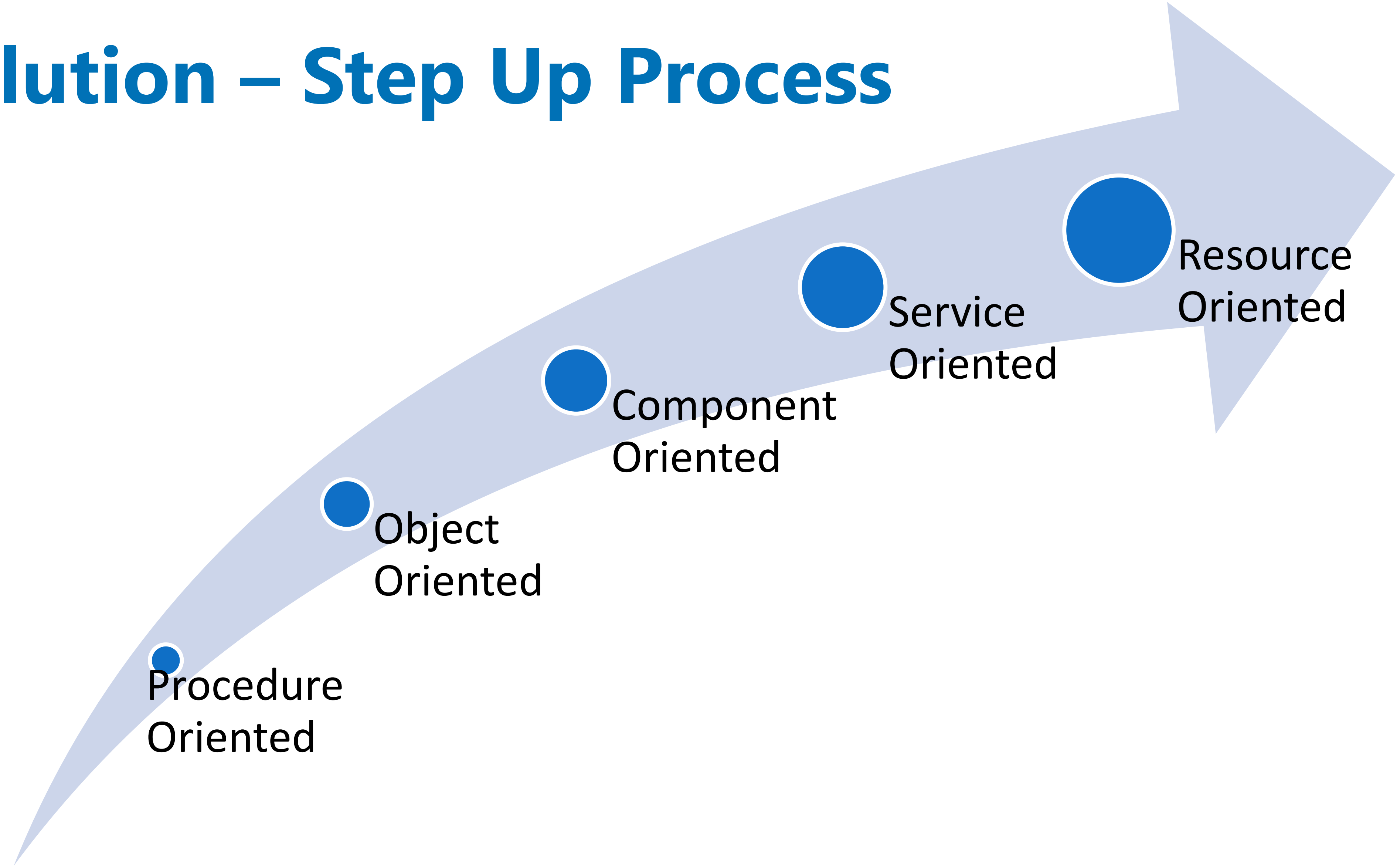
B Nagaraju

http://nbende.wordpress.com

Agenda

- Evolution of Programming
- OOP using C#
- C# Classes and Objects
- Demos

Evolution – Step Up Process



Procedure/Object-Oriented Programming

- **Variables** are named computer memory locations used to hold values that may vary
- **Operations** are usually **called** or **invoked** to manipulate variables
- A **procedural program** defines the variable memory locations, then calls a series of procedures to input, manipulate, and output the value stored in those locations
- A single procedural program often contains hundreds of variables and thousands of procedure calls

Object-Oriented Programming

- **Object-oriented** programming is an extension of procedural programming, which in addition to variables and procedures contains: objects, classes, encapsulation, interfaces, polymorphism, and inheritance
- **Objects** are object-oriented components
- **Attributes** of an object represent its characteristics
- A **class** is a category of objects or a type of object
- An **instance** refers to an object based on a class

Object-Oriented Programming

- **A Simple Example**
  - An Automobile is a class whose objects have the following attributes: year, make, model, color, and current running status
  - Your 1997 red Chevrolet is an instance of the class that is made up of all Automobiles
- **Methods** of classes are used to change attributes and discover values of attributes
  - The Automobile class may have the following methods: getGas(), accelerate(), applyBreaks()

Object-Oriented Programming

- Methods and variables in object-oriented programming are **encapsulated,** that is, users are only required to understand the **interface** and not the internal workings of the class
- **Polymorphism** and **Inheritance** are two distinguishing features in the object-oriented programming approach
- Polymorphism describes the ability to create methods that act appropriately depending on the context
- Inheritance provides the ability to extend a class so as to create a more specific class

Benefits of Object-Oriented Programming?

	Structured Design Tendencies		Object-Oriented Tendencies
1	Process-Centered	↔	Object-Centered
2	Reveals Data	↔	Hides Data
3	Single Unit	↔	Modular Units
4	One-Time Use	↔	Reusable
5	Ordered Algorithm	↔	Nonordered Algorithm

The C# Programming Language

- C# was developed as an object-oriented and component-oriented language
- C# (like Java) is modeled after the C++ programming language
- C# satisfies more object oriented rules than C++….
- C# does NOT require the use of object destructors, forward declarations, or #include files
- …..

Agenda

- Creating and Using classes
- What is a Class
- Creating Objects
- Accessing Class Members
- Demos

- Visual C# is an object-oriented programming language. All of the logic for a C# application is contained in classes and structs.
- A Class is a blueprint from which you can create objects
- A Class defines the characteristics of an object.

```
 class <classname>
 {
 }
```

- An object is an instance of class

Key Points

When you create a C# application, you use classes that represent the principal data types in your application. The .NET Framework provides a large number of reusable utility classes, but you can also define your own classes that encapsulate data and logic that is specific to your own applications.

What Is a Class?

You can think of a class as a blueprint from which you can create objects. A class

Note:
The term *instance* is often used as an alternative to *object*.

What Is an Object? An object is an instance of a class. If a class is like a blueprint, an object is an item that you create by using that blueprint. The class is the definition of an item; the object is the item itself.

- Creating a Class Demo

Adding Members to classes

- Members define the data and behaviour of the class
  - Fields
  - Methods
- **Key Points**
  - You can add fields and methods to a class that define the data and behavior of that class.
  - You can define any number of fields and methods in a class, depending on the purpose and intended functionality of the class.

Defining Fields

- You can think of a field as a variable that is scoped to the class.
- All methods that are defined in the class can access the field. Like a variable, each field has a name, a data type, and an access modifier.
- If you do not explicitly specify an access modifier for a field, the default access level is private, which means that it can be accessed only by methods that are defined in the class.
- If you want to make the field available to methods that are defined in other classes, you can mark the field as public.

Defining Methods

- A method is a procedure or function inside a class. You use methods to implement the behavior of a class.
- Each method has a name, a parameter list, a return type, and an access modifier
- A method has complete and unrestricted access to all of the other members in the class. This is an important aspect of object-oriented programming
- Methods encapsulate operations on the fields in the class.

- Creating a Class with Members i.e Fields & Methods

Demo

Creating Objects

- Objects are initially unassigned
- Before you can use a class, you must create an instance of that class
- You can create a new instance of a class by using the **new** operator
  ex: <class> <objname> = new <class>();
- The new operator does 2 things
  - Causes the CLR to allocate memory for the object
  - Invokes a constructor to initialize the object

- Creating Objects for prepared classes

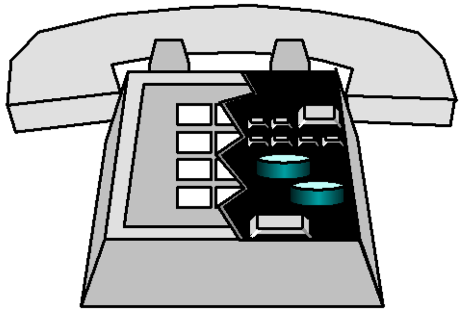
Demo

Accessibility and Scope

- Access modifiers are used to define the accessibility level of class members

Declaration	Definition
public	Access not limited. It can be used in the current and other assemblies.
private	Access limited to the containing class i.e. to all the statements of current class.
internal	Access limited to the current assembly only. Not accessible to other assemblies.
protected	Access limited to the containing class and to types derived from the containing class
protected internal	Access limited to the containing class, derived classes, or to members of this program

What Is Encapsulation?



- The first pillar of OOP is encapsulation .
- Hiding internal details
- Makes your object easy to use
- language’s ability to hide unnecessary implementation details from the object user
- Grouping related pieces of information and processes into self-contained unit
- Makes it easy to change the way things work under the cover without changing the way users interact
- Properties in C#(Component Based) Programming provide encapsulation.

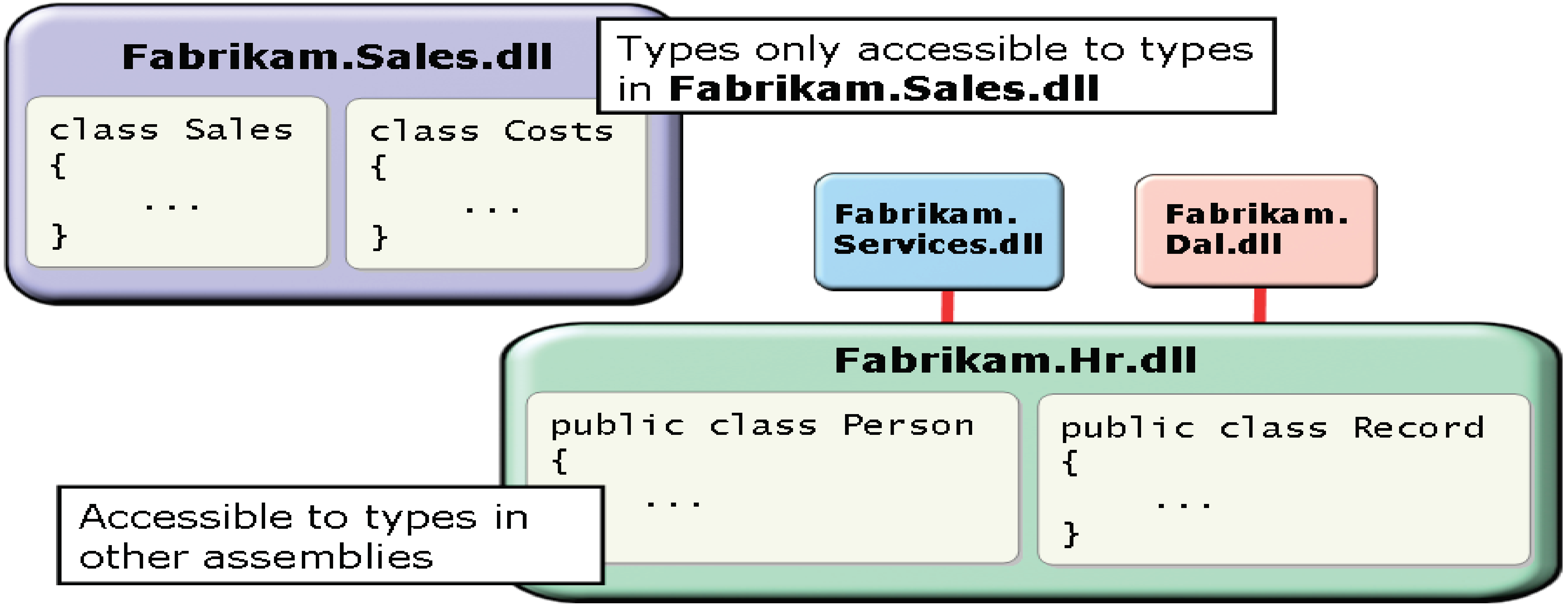
- Encapsulation provides a way to preserve the integrity of an object’s state data
- Rather than defining public fields (which can lead to data corruption), you should get in the habit of defining private data, which is indirectly manipulated using one of two main techniques.

1. *You can define a pair of public accessor (get) and mutator (set) methods.*
2. *You can define a public .NET property.*

Internal is the default access modifier for types

Internal permits access only to types in the same assembly

Public permits access to other types in other assemblies


Fabrikam.Sales.dll
Types only accessible to types in Fabrikam.Sales.dll
Fabrikam.Services.dll
Fabrikam.Dal.dll
Fabrikam.Hr.dll
Accessible to types in other assemblies

```
class Sales
{
    ...
}
```

```
class Costs
{
    ...
}
```

```
public class Person
{
    ...
```

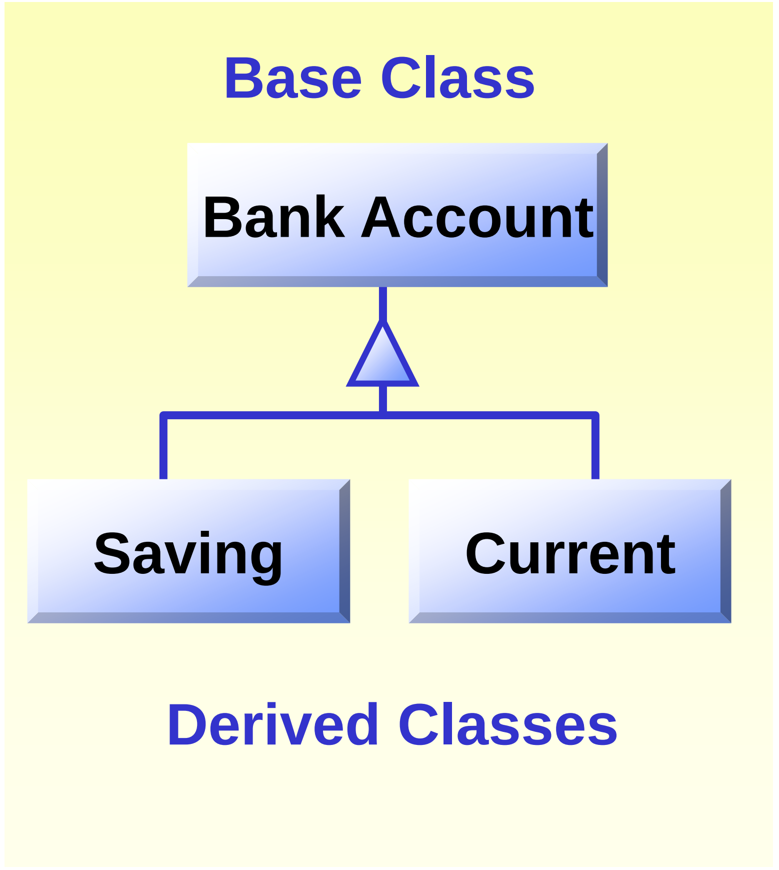
```
public class Record
{
    ...
}
```

Inheritance

- What is Inheritance
- The .NET framework Inheritance Hierarchy
- Demo
- abstract and virtual methods

What Is Inheritance?

- Inheritance specifies an is-a-kind-of relationship (UML)
- Derived classes inherit properties and methods from a base class, allowing code reuse
- Derived classes become more specialized
- We can use inheritance to develop better code faster, and with fewer bugs
- Developing an object hierarchy is an important process



- The object hierarchy should be well designed, and we should avoid code duplication. Understanding inheritance in the .NET Framework is fundamental to this process
- Inheritance is a key concept in the world of object orientation. You can use inheritance as a tool to avoid repetition when you are defining different classes that have several features in common and are related to each other.
- C# supports single inheritance only. You cannot define a class that directly inherits from more than one base class.

Inheritance enables you to define new types based on existing types:

- For example, **Manager** and **ManualWorker** classes might inherit from an **Employee** class
- Fields and methods in the **Employee** class are inherited by **Manager** and **ManualWorker**
- **Manager** and **ManualWorker** can define their own fields and behavior
- Define accessible members as **protected** in the base class

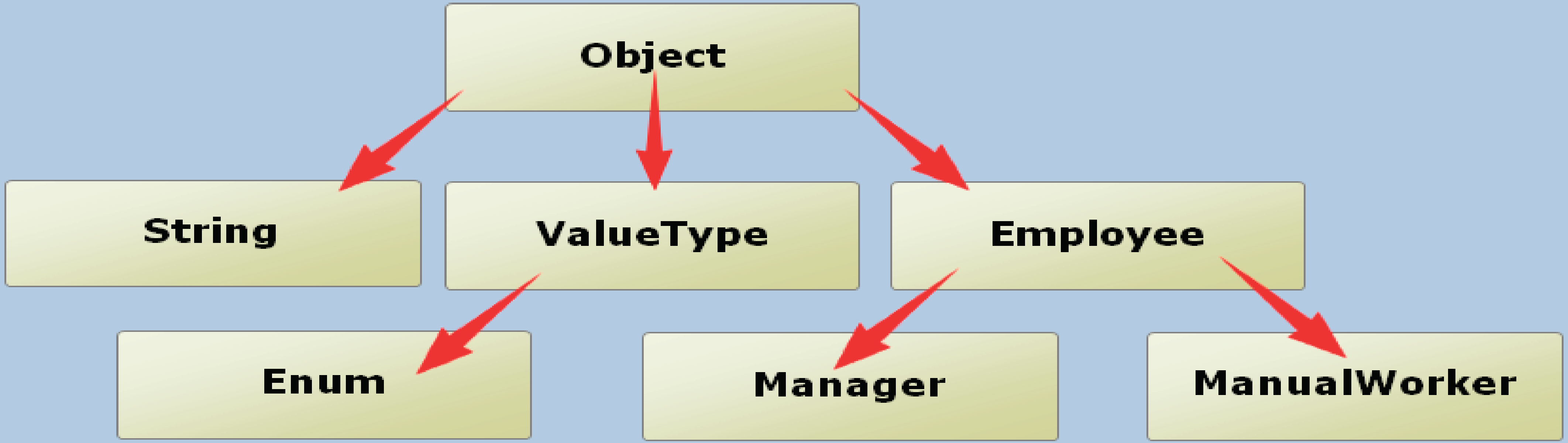
```
// Base class
class Employee
{
    protected string empNum;
    protected string empName;
    protected void DoWork()
    { ... }
}

// Inheriting classes
class Manager : Employee
{
    public void DoManagementWork()
    { ... }
}

class ManualWorker : Employee
{
    public void DoManualWork()
    { ... }
}
```

All types inherit directly or indirectly from the **System.Object** class

- No need to specify **: Object** in the class definition
- Structs and enums inherit from **ValueType**



You cannot define your own inheritance hierarchy by using structs and enums

Demo

Abstract and Sealed Classes

- C# supplies another keyword, sealed, that prevents inheritance from occurring. When you mark a class as sealed, the compiler will not allow you to derive from this type.
- The abstract keyword on other hand enables you to create classes and class members that are incomplete and must be implemented in a derived class
- Abstract class may contain abstract and non-abstract function members

```
public abstract class A
{ // Class members
here. }
```

- An abstract class cannot be instantiated – useful in team development.
- The purpose of an abstract class is to provide a common definition of a base class that multiple derived classes can share.
- For example, a class library may define an abstract class that is used as a parameter to many of its functions, and require programmers using that library to provide their own implementation of the class by creating a derived class.
- Abstract classes may also define abstract methods. This is accomplished by adding the keyword abstract before the return type of the method. For example:

```
public abstract class A
{
public abstract void DoWork(int i);
 }
```

abstract and virtual Methods

- If a base class declares a member as abstract, that method must be overridden in any non-abstract class that directly inherits from that class.
- When a base class declares a method as virtual, a derived class can override the method with its own implementation using override.
- If a derived class is itself abstract, it inherits abstract members without implementing them.
- Abstract and virtual members are the basis for polymorphism, which is the second primary characteristic of object-oriented programming.
- Abstract methods have no implementation, so the method definition is followed by a semicolon instead of a normal method block

```
public abstract class A
{
 public abstract void DoWork(int i);
 }
```

- Derived classes of the abstract class must implement all abstract methods.
- Classes can be declared as sealed by putting the keyword sealed before the class definition.

```
public sealed class D
{ // Class members here. }
```

- A sealed class cannot be used as a base class. For this reason, it cannot also be an abstract class. Sealed classes prevent derivation. Because they can never be used as a base class, some run-time optimizations can make calling sealed class members slightly faster

- A member on a derived class that is overriding a virtual member of the base class can declare that member as sealed. This negates the virtual aspect of the member for any further derived class. This is accomplished by putting the sealed keyword before the override keyword in the class member declaration. For example:

```
public class D : C
{
    public sealed override void DoWork() { } // further no overriding
}
```

this & base keywords of C#

this

- The this keyword is a predefined variable available in non-static function members
  - Used to access data and function members unambiguously

name is a parameter and a field.

```
public class Person {
  private string name;
  public Person(string name) {
    this.name = name;
  }
  public void Introduce(Person p) {
    if (p != this)
      Console.WriteLine(“Hi, I’m “ + name);
  }
}
```

base

- The base keyword can be used to access class members that are hidden by similarly named members of the current class

```
public class Shape {
  private int x, y;
  public override string ToString() {
    return "x=" + x + ",y=" + y;
  }
}
internal class Circle : Shape {
  private int r;
  public override string ToString() {
    return base.ToString() + ",r=" + r;
  }
}
```

Constructors

- Constructors are special methods designed to initialize fields
  - CLR executes constructor whenever new is called on a class
  - creatable classes must have at least one constructor
  - me name as class name
  - aded
  -

```
public class Customer
{
  private string m_Name;
  private string m_Phone;
  public Customer(string Name, string Phone)
  {
    m_Name = Name;
    m_Phone = Phone;
  }
}
```

```
Customer c1;
c1 = New Customer("Wendy", "432-4636");
Customer c2 = New Customer("Bob", "555-1212");
```

Default constructor

- Constructors with no parameter are called as "default constructor"(Parameterless constructor)
  - allows client to create object without passing parameters
  - C# compiler automatically adds a default constructor to classes that have no explicit constructor
  - default constructor (if desired) must be explicitly added to classes that contain other constructors

```
public class Class1{
  //no explicit constructor
}
```

```
public class Class2{
 public Class2(string s){
  //implementation
 }
}
```

```
public class Class3{
 public Class3(string s){
    //implementation
  }
 public Class3(){
   //implementation
  }
}
```

```
Class1 c1 = new Class1();
Class2 c2 = new Class2();
Class3 c3 = new Class3();
```

Calls default constructor

Illegal: no default constructor

Calls default constructor

Classes and Objects[Constructors]

```
public class Time
{
        public void DisplayCurrentTime( )
        {
                System.Console.WriteLine("{0}/{1}/{2} {3}:{4}:{5}",Month, Date, Year, Hour, Minute, Second);
        }
        public Time(System.DateTime dt)
        {
                Year = dt.Year;
                Month = dt.Month;
                Date = dt.Day;
                Hour = dt.Hour;
                Minute = dt.Minute;
                Second = dt.Second;
        }
        int Year; int Month; int Date; int Hour; int Minute; int Second;
}
public class Tester
{
        static void Main( )
        {
                System.DateTime currentTime = System.DateTime.Now;
                Time t = new Time(currentTime);
                t.DisplayCurrentTime( );
        }
}
```

Overloading methods and properties

- Two or more class members can be given the same name
  - you can overload a set of methods
  - Overloaded members must differ with their parameter lists
  - parameter lists must differ in size and/or in sequence of types
  - r name

```
public class CustomerManager
{
  public string GetCustomerInfo(int ID)
  {
    //implementation
  }
  public string GetCustomerInfo(string Name)
  {
    //implementation
  }
}
```

```
//client-side code
string info1, info2;
CustomerManager mgr = new
CustomerManager();
//call GetCustomerInfo(Integer)
info1 = mgr.GetCustomerInfo(23);
//call GetCustomerInfo(String)
info2 = mgr.GetCustomerInfo(“fu");
```

Base classes and constructors

- Constructors and base types have "issues"
  - derived class contract doesn't include base class constructors

```
public class Human
{
  protected string m_Name;
  public Human(string Name)
  {
    //implicit call to System.Object constructor
    m_Name = Name;
  }
}
public class Programmer : Human
{
  //doesn't compile
  //base class has no accessible default constructor!
}
```

Constructor Chaining

```
 public class a {
          public a()
          {
                    System.Console.WriteLine(“a Class");
          }
}

public class b : a {
          public b()
          {
                    System.Console.WriteLine("b Class");
          }
}
public class c : b {
          public c()
          {
                    System.Console.WriteLine(“c Class");
          }
}
public class d : c {
          public d()
          {
                    System.Console.WriteLine("d Class");
          }
}
```

Calling base class constructor-1

– Base cla

  • can c

```
public class Human
{
  public Human()
    {


    }
  protected string m_Name;
  public Human(string Name)
  {
    //implicit call to System.Object constructor
    m_Name = Name;
  }
}
public class Programmer : Human
{


}
```

default Constructor

Calling base class constructor-2

– Base class cor
  • can only be

```
public class Human
{
  protected string m_Name;
  public Human(string Name)
  {
    //implicit call to System.Object constructor
    m_Name = Name;
  }
}
public class Programmer : Human
{
  public Programmer(string Name):base(Name)
    {
    //chaining a call to base class constructor
    }
}
```

Special syntax for constructors

Polymorphism

- The word polymorphism can be broken down into two different words, ‘poly’ meaning many and ‘morph’ meaning forms, and hence the meaning ‘having many forms’.
- polymorphism is used to imply one name with multiple functionality
- In polymorphism we declare methods with the same name and different parameters in same class or methods with the same name and same parameters in different classes. Polymorphism has the ability to provide different implementation of methods that are implemented with the same name.
- 2 types of polymorphism we have
  - Static Polymorphism
  - Dynamic Polymorphism

static polymorphism

- When the response to a function is determined at the time of compiling then it is called as static polymorphism hence it is als phism
- In this polymo ded with the same nam res. Hence it is also calle

```
public class addition
{
 public int Add(int a, int b)
  {
 return a+b;
 }
public double Add(int x, int y, int z)
{
return x+y+z;
}
}
```

Overloading methods and properties

- Two or more class members can be given the same name
  - you can overload a set of methods
  - Overloaded members must differ with their parameter lists
  - parameter lists must differ in size and/or in sequence of types

  r name

```
public class CustomerManager
{
  public string GetCustomerInfo(int ID)
  {
    //implementation
  }
  public string GetCustomerInfo(string Name)
  {
    //implementation
  }
}
```

```
//client-side code
string info1, info2;
CustomerManager mgr = new
CustomerManager();
//call GetCustomerInfo(Integer)
info1 = mgr.GetCustomerInfo(23);
//call GetCustomerInfo(String)
info2 = mgr.GetCustomerInfo(“fu");
```

Dynamic Polymorphism

- Response to a function is determined at run time and hence is known as run time polymorphism
- In this polymorphism, the different methods have the same name and also the same signature but differ in the implementation.
- In this polymorphism the methods are overridden and hence it is also known as method overriding
- The 'virtual' and 'override' keywords are used for method overriding

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*.**

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()
    {}}

Interfaces in C#

Interfaces

- An interface defines a contract
  - An interface is a type
  - Includes methods, properties, indexers, events
  - Any class or struct implementing an interface must support all parts of the contract
- Interfaces provide no implementation
  - When a class or struct implements an interface it must provide the implementation
- Interfaces provide polymorphism
  - Many classes and structs may implement a particular interface

- Interfaces specify the public services (methods and properties) that classes must implement
- Interfaces vs. abstract classes w.r.t default implementations
  - Interfaces provide no default implementations
  - Abstract classes may provide some default implementations
    - If no default implementations can/are defined – do *not* use an abstract class, use an interface instead
- Interfaces are used to "bring together" or relate to each other disparate objects that relate to one another only through the interface
  - I.e., provide uniform set of methods and properties for disparate objects
  - E.g.: A person and a tree are disparate objects

    Interface can define age and name for these disparate objects
  - Enables polymorphic processing of age and name for person & tree objects

Interfaces Example

```
public interface IDelete {
  void Delete();
}
public class TextBox : IDelete {
  public void Delete() { ... }
}
public class Car : IDelete {
  public void Delete() { ... }
}
```

```
TextBox tb = new TextBox();
IDelete iDel = tb;
iDel.Delete();

Car c = new Car();
iDel = c;
iDel.Delete();
```

Interfaces

Multiple Inheritance

- Classes and structs can inherit from multiple interfaces
- Interfaces can inherit from multiple interfaces

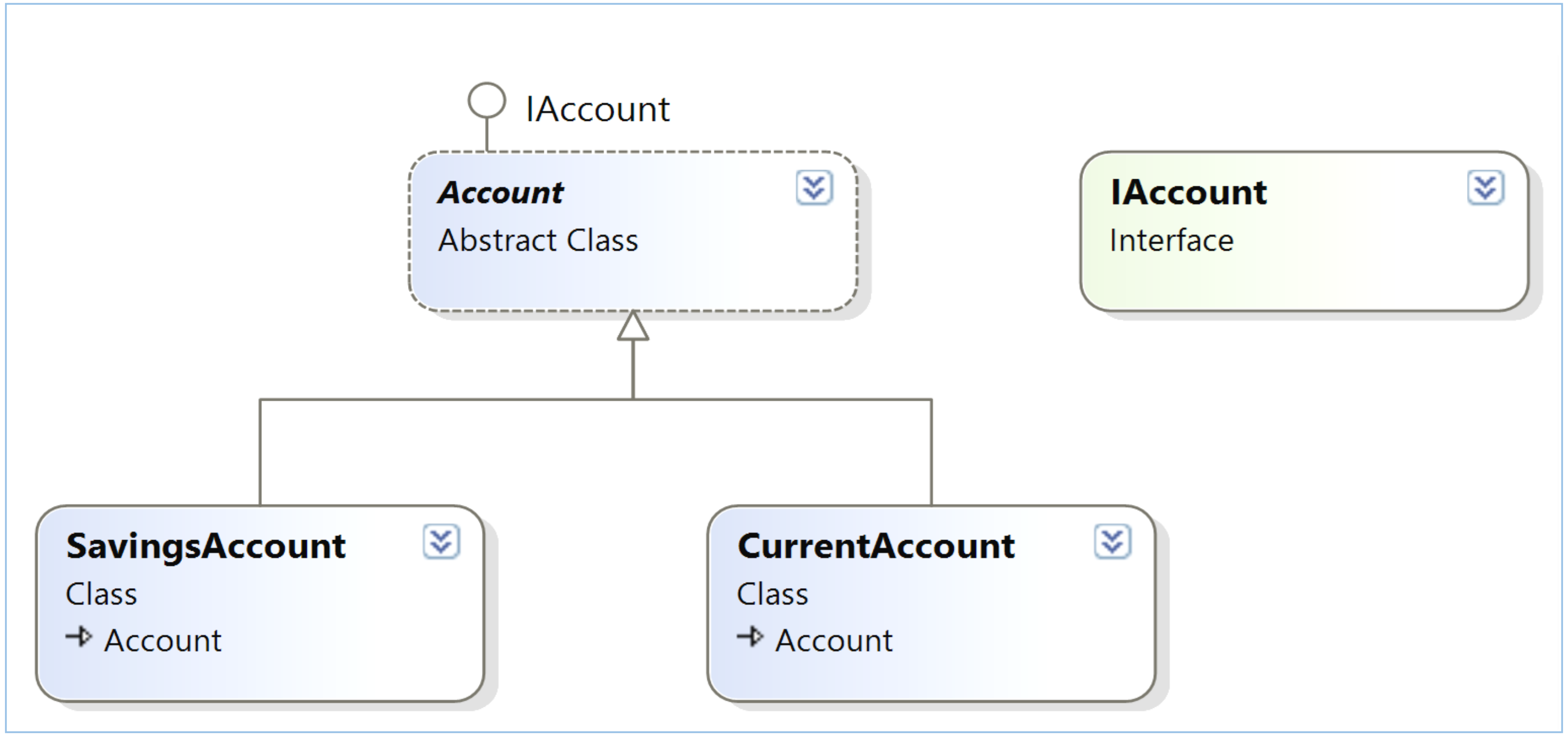
```
interface IControl {
  void Paint();
}
interface IListBox: IControl {
  void SetItems(string[] items);
}
interface IComboBox: ITextBox, IListBox {
}
```

Interfaces
Explicit Interface Members

- If two interfaces have the same method name, you can explicitly specify disambiguate the

```
interface IControl {
  void Delete();
}
interface IListBox: IControl {
  void Delete();
}
interface IComboBox: ITextBox, IListBox {
  void IControl.Delete();
  void IListBox.Delete();
}
```

Example


IAccount
Account
Abstract Class
IAccount
Interface
SavingsAccount
Class
Account
CurrentAccount
Class
Account

```
interface IAccount
  {
      int AccountNo { get; set; }
      string AccountName { get; set; }
      int Balance { get; }
      int Amount { get; set; }
      void Deposit();
      void WithDraw();
  }
```

Dynamic Binding

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

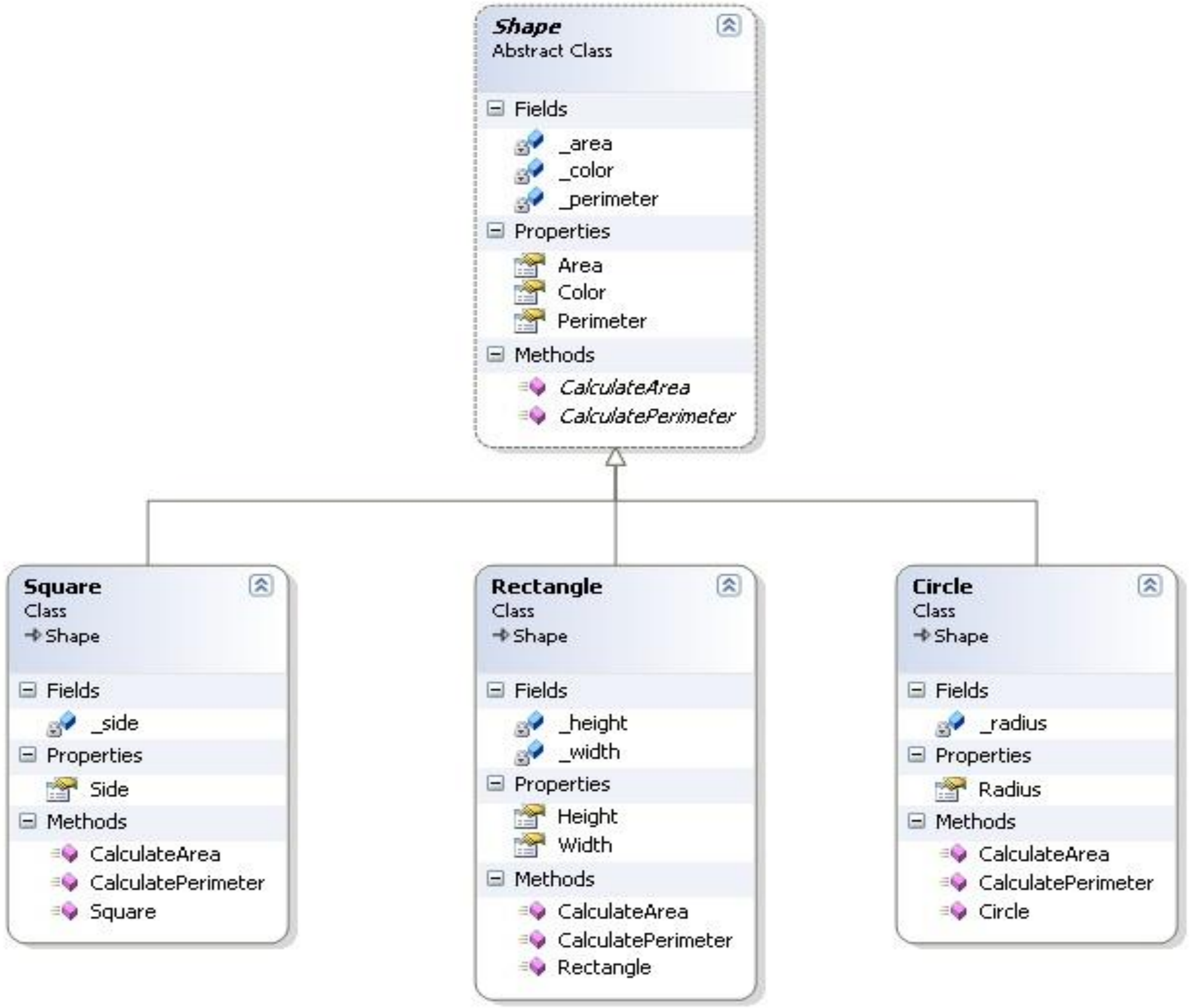
```
day.Celebrate();
```

- 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

Certification Questions:

What accessor should you use to make class members accessible to child classes?

What types inherit from the **Object** class?



Shape
Abstract Class
Fields
_area
_color
_perimeter
Properties
Area
Color
Perimeter
Methods
CalculateArea
CalculatePerimeter
Square
Class
Shape
Fields
_side
Properties
Side
Methods
CalculateArea
CalculatePerimeter
Square
Rectangle
Class
Shape
Fields
_height
_width
Properties
Height
Width
Methods
CalculateArea
CalculatePerimeter
Rectangle
Circle
Class
Shape
Fields
_radius
Properties
Radius
Methods
CalculateArea
CalculatePerimeter
Circle

How to Create a Derived Class

```
public class Bank_Account
{           protected string AccountNo;
            protected double BalanceAmount;

            public bool Deposit(string AccountNo,double Amount)
             { }
            public bool Withdraw(string AccountNo,double Amount)
             { }
            public double GetBalance(string  AccountNo)
            { }
}
```

(contd…)

```
public class Saving : Bank_Account{

protected double InterestRate;
public void CalculateInterest(string AccountNo) { }
}

public class Current : Bank_Account{

protected double InterestRate;
public void CalcInterest(string AccountNo) { }
}


…
Saving IBMAccount=new Saving();
IBMAccount.Deposit(“x12345”,200000.50);
```