Inheritance and Polymorphism

Introduction

- Inheritance is a form of software reuse in which a new class is created by absorbing an existing class's members and enhancing them with new or modified capabilities.
- Inheritance lets you save time during app development by reusing proven, high-performance and debugged high-quality code.
- This also increases the likelihood that a system will be implemented effectively.

Introduction

- The existing class from which a new class inherits members is called the base class or parent class.
- And the new class is the derived class or child class.
- Each derived class can become the base class for future derived classes.
- A derived class normally adds its own fields, properties and methods.
- Therefore, it's more specific than its base class.

Types of Inheritance

- **Single Inheritance:** With single inheritance, one class can derive from one base class. This is a possible scenario with C#.
- **Multiple Inheritance:** Multiple inheritance allows deriving from multiple base classes. C# does not support multiple inheritance with classes, but it allows multiple inheritance with interfaces.
- **Multilevel Inheritance:** Multilevel inheritance allows inheritance across a bigger hierarchy. Class B derives from class A, and class C derives from class B. This is supported and often used with C#.
- **Interface inheritance:** Interface inheritance defines inheritance with interfaces. Here, multiple inheritance is possible.

Inheritance Syntax

- If you want to declare that a class derives from another class, use this syntax.

```
class MyDerivedClass : MyBaseClass
{
   // data members and methods here
}
```

- If you do not specify a base class in a class definition, the C# compiler will assume that `System.Object` is the base class.

```
class MyClass : object // derives from
                       // System.Object
{
   // etc.
}
```

- Hence, these two pieces of code yield the same result.

```
class MyClass      // derives from
                   // System.Object
{
   // etc.
}
```

- For the sake of simplicity, the second form is more common.

Introduction

- Every class directly or indirectly extends (or “inherits from”) `object` - a C# keyword that’s an alias for `System.Object`.
- Next slide lists class `object`’s methods, which every other class inherits.
- In single inheritance, a class is derived from one direct base class.
- C# supports only single inheritance.
- You can use interfaces to achieve multiple inheritance.

Class `System.Object`'s Methods

Name	Description
`Equals()`	Determines whether the specified object is equal to another object.
`GetHashCode()`	Serves as the default hash function.
`ToString()`	Returns a string that represents the current object.

- Link:
- https://docs.microsoft.com/en-us/dotnet/api/system.object?view=netframework-4.8

Base Classes and Derived Classes

- Often, an object of one class is an object of another class as well.
- For example, in geometry, a rectangle is a quadrilateral.
  - And so are squares, parallelograms and trapezoids.
- Thus, class `Rectangle` can be said to inherit from class `Quadrilateral`.
- In this context, class `Quadrilateral` is a base class and class `Rectangle` is a derived class.
- A rectangle is a specific type of quadrilateral, but it's incorrect to claim that every quadrilateral is a rectangle – the quadrilateral could be a parallelogram or some other shape.

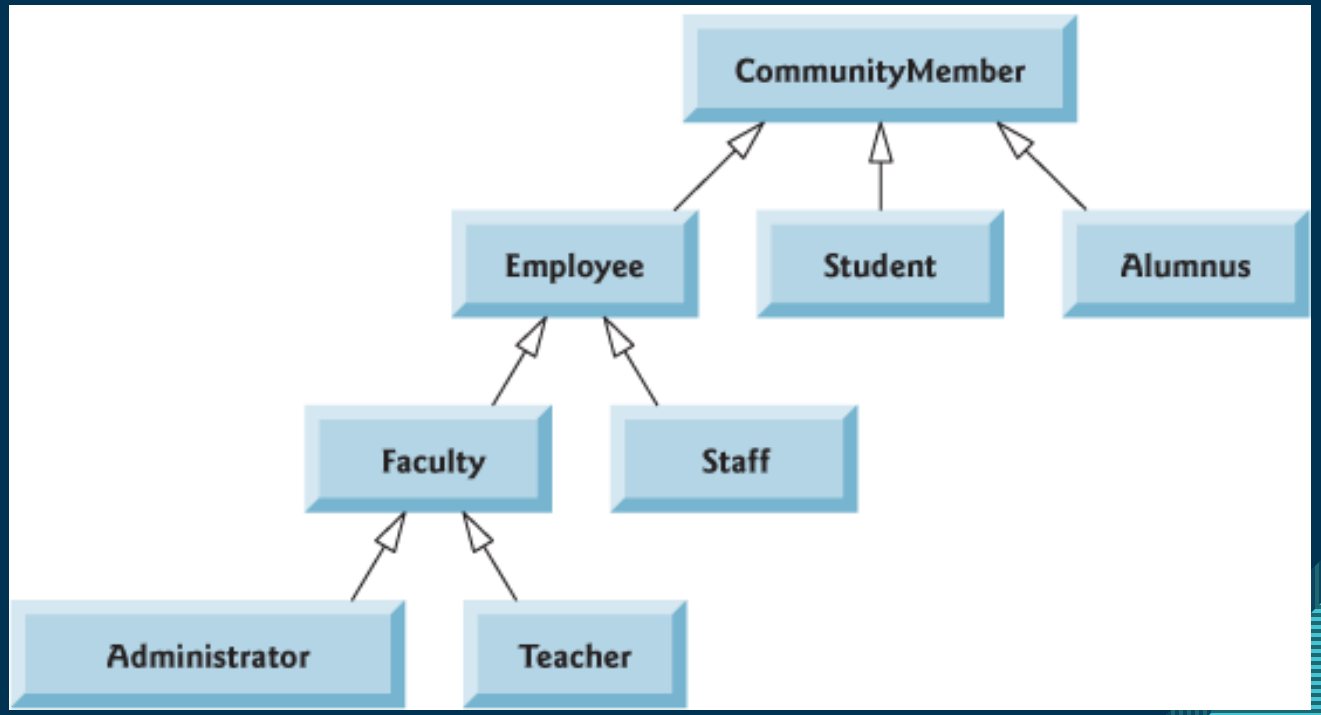
- *Base classes tend to be more general, and derived classes tend to be more specific.*

Inheritance Examples

Base Class	Derived Class
Student	GraduateStudent, UndergraduateStudent
Shape	Circle, Triangle, Rectangle
Loan	CarLoan, HomeImprovementLoan, MortgageLoan
Employee	Faculty, Staff, HourlyWorker, CommissionWorker
SpaceObject	Star, Moon, Planet, FlyingSaucer
BankAccount	CheckingAccount, SavingsAccount

Base Classes and Derived Classes

- Inheritance relationships form treelike hierarchical structures.
- A base class exists in a hierarchical relationship with its derived classes.
- For example, employees are either faculty members or staff members.
- Faculty members are either administrators (such as deans and department chairpersons) or teachers.
- The hierarchy could contain many other classes.



`protected` Members

- A class's `public` members are accessible wherever the app has a reference to an object of that class.
- A class's `private` members are accessible only within the class itself.
- A base class's `private` members are inherited by its derived classes but are not directly accessible by derived-class methods and properties.
- Using `protected` access *offers an intermediate level of access* between `public` and `private`.
- A base class's `protected` members can be accessed by members of that base class and by members of its derived classes, but not by clients of the class.
- Derived-class methods can refer to `public` and `protected` members inherited from the base class simply by using the member names.

Constructor Calls in Inheritance

- Instantiating a derived-class object begins a chain of constructor calls.
- The derived-class constructor, before performing its own tasks, invokes its direct base class's constructor.
  - Either implicitly (calling the base class's default constructor or parameterless constructor) or explicitly (via a constructor initializer with the `base` reference).

- Similarly, if the base class is derived from another class (as every class except `object` is), the base-class constructor invokes the constructor of the next class up in the hierarchy, and so on.

Constructor Calls in Inheritance

```
public class BaseClass
{
    public BaseClass()
    { Console.WriteLine("Base class constructor called"); }
}

public class DerivedClass : BaseClass
{
    public DerivedClass()
    { Console.WriteLine("Derived class constructor called"); }
}

class Program
{
    static void Main(string[] args)
    {
        // instantiating derived class
        DerivedClass derived = new DerivedClass();
    }
}
```

Output:

```
Base class constructor called
Derived class constructor called
```

Constructor Calls in Inheritance

- Let's assume the base class has 2 constructors:
  - One parameterless
  - Another that accepts a parameter.
- We saw in the previous example that the derived class implicitly calls the parameterless constructor.
- If we want to call a different constructor (maybe the one that accepts parameter), then we have to make use of the `base` keyword.
- Example on the next slide.

Constructor Calls in Inheritance

```
public class BaseClass {
    public BaseClass()
    { Console.WriteLine("Base class constructor called"); }

    public BaseClass(string message)
    { Console.WriteLine("Constructor with message: " + message); }
}

public class DerivedClass : BaseClass {
    public DerivedClass() : base("Hello from derived class")
    { Console.WriteLine("Derived class constructor called"); }
}

class Program {
    static void Main(string[] args)
    {
        // instantiating derived class
        DerivedClass derived = new DerivedClass();
    }
}
```

Output:

```
Constructor with message: Hello from derived class
Derived class constructor called
```

Exercise

- Consider an inheritance hierarchy containing types of employees in a company's payroll app to discuss the relationship between a base class and a derived class.

- In this company:
  - Commission Employees: Who will be represented as objects of a base class – are paid a percentage of their sales.

  - Base-Salary plus Commission Employees: Who will be represented as objects of a derived class – receive a base salary plus a percentage of their sales.

Exercise - *Continued*

- Class `CommissionEmployee` declares `private` fields `_id`, `_name`, `_grossSales` and `_commissionRate`.
- Provides `public` properties `Id`, `Name`, `GrossSales` and `CommissionRate` for manipulating these fields.
- Constructor that accepts parameters to initialize the properties.
- Method `Earnings` to calculate the employee's earnings.
- Method `ToString` to print the employee's information.

Exercise - *Continued*

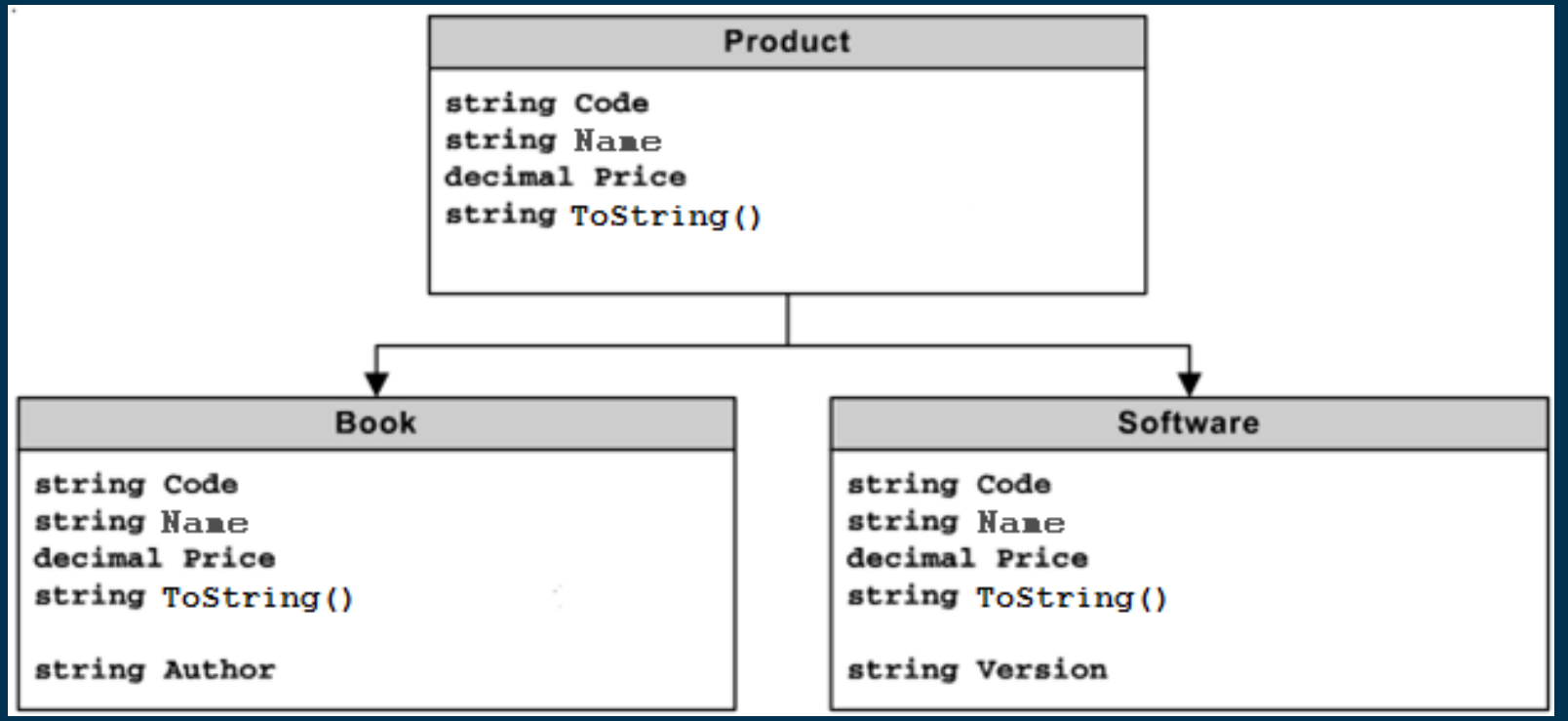
- Class `SalaryPlusCommissionEmployee` inherits from `CommissionEmployee`.
- Therefore, it also inherits the `public` properties such as `Id`, `Name`, `GrossSales` and `CommissionRate` of `CommissionEmployee`.
- It only needs to declare `private` field `_salary` and `public` property `Salary`.
- Constructor that accepts parameters to initialize all the five properties.
- Out of those 5 properties, 4 are passed over to the base class constructor.
  - Remember that base class doesn't have a parameterless constructor.
  - So, we have to call the constructor with parameters.
- Method `Earnings` to calculate the employee's earnings.
  - This method further calls the base class `Earnings` method to avoid the duplicate code.
- Method `ToString` to print the employee's information.
  - It also calls the base class `ToString` method.

Do It Yourself!

- Create a class named `Shape` with properties named `Height`, `Width`, and a method `Area`.
- Add two classes that derive from it – `Rectangle` and `Square`.
- Add any additional members you feel are appropriate and implement the `Area` method correctly.

Do It Yourself!

- Create a class hierarchy that looks like this.
- Then, test the `Book` and `Software` classes in the `main` method.



Polymorphism

Introduction to Polymorphism

- Polymorphism means one name many forms.
- One object behaving as multiple forms.
- In other words, "Many forms of a single object is called Polymorphism.“

- Polymorphism enables you to write apps that process objects that share the same base class in a class hierarchy as if they were all objects of the base class.

Real World Example of Polymorphism

- **Example 1:**
  - A student's behavior towards the teacher.
  - A student's behavior towards friends.
  - Here student is an object, but the attitude is different in different situations.

- **Example 2:**
  - Person behaves as a son/daughter in the house.
  - And, that person behaves like an employee in the office.

- With polymorphism, the same method or property can perform different actions depending on the run-time type of the instance that invokes it.

Polymorphism Example

- As an example, suppose we design a video game that manipulates objects of many different types, including objects of classes `Martian`, `Venusian`, `Plutonian`, `SpaceShip` and `LaserBeam`.
- Each class inherits from the common base class `SpaceObject`, which contains method `Draw`.
- Our app maintains a collection (e.g., a `SpaceObject` array) of references to objects of the various classes.
- To refresh the screen, the app periodically sends each object the same message – namely, `Draw`, while object responds in a unique way.

- The same message (in this case, `Draw`) sent to a variety of objects has *many forms* of results.

Apps Are Easy to Extend

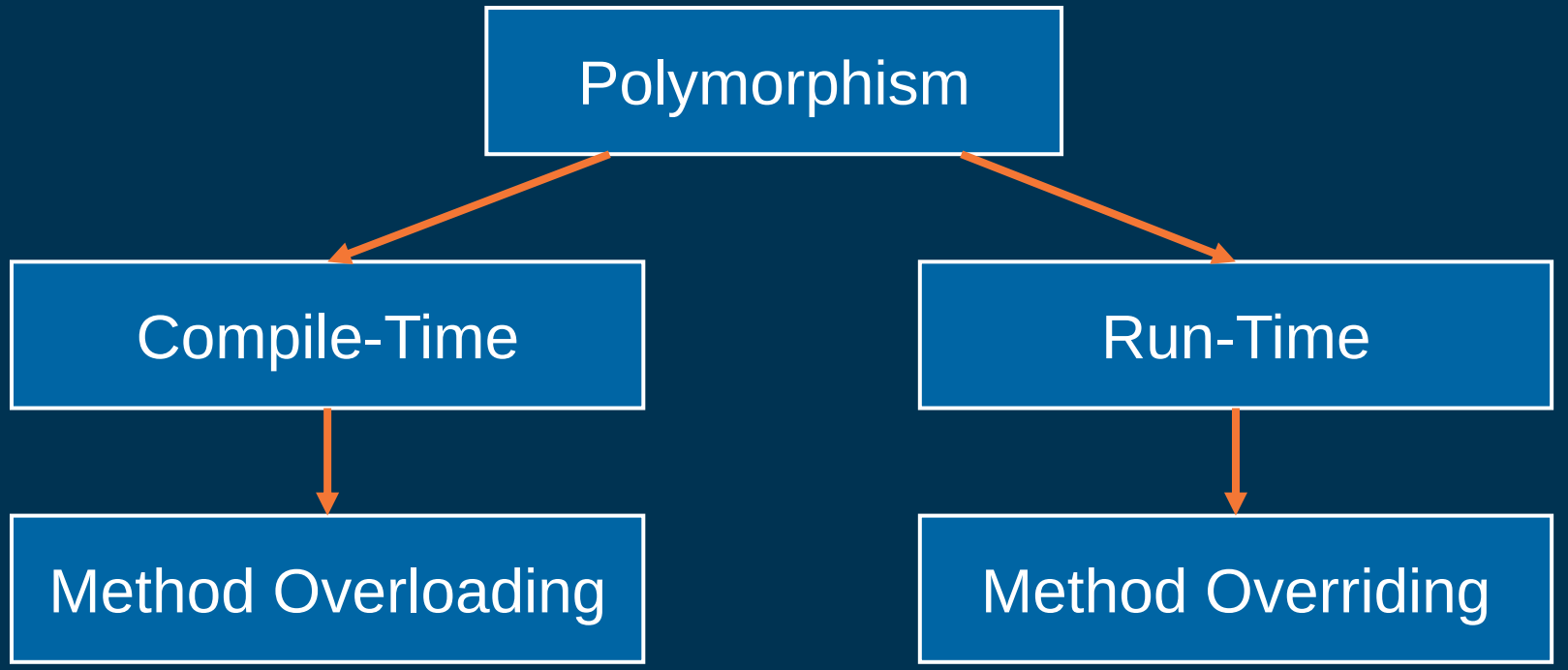
- With polymorphism, we can design and implement apps that are easily *extensible*.
- New classes can be added with little or no modification to the general portions of the app, as long as the new classes are part of the inheritance hierarchy.
- The only parts of an app that must be altered to accommodate new classes are those that require direct knowledge of the new classes that you add to the hierarchy.

Apps Are Easy to Extend

- For example, we want to add `Mercurian` objects to our video game.
- To do so, we must build a `Mercurian` class that extends `SpaceObject` and provides its own `Draw` method.
- When objects of class `Mercurian` appear in the `SpaceObject` collection, the app invokes method `Draw`, exactly as it does for every other object in the collection, regardless of its type, so the new `Mercurian` objects simply “plug right in” without much modifications.
- Thus, without modifying the app (other than to build new classes and modify the code that creates new objects), you can use polymorphism to include additional types that might not have been envisioned when the app was created.

Types of Polymorphism

- Theoretically, there are two types of polymorphism:
  - Static or compile time polymorphism
  - Dynamic or runtime polymorphism



Static or Compile Time Polymorphism

- In static polymorphism, the *decision is made at compile-time*.
- Which method is to be called is decided at compile-time.
- *Method overloading is an example* of this.

- Method overloading is a concept where a class can have more than one method with the same name and different parameters.
- Compiler checks the type and number of parameters passed on to the method and decides which method to call at compile time.
- It will give an error if there are no methods that match the method signature.

Example

```
public class MethodOverloadingExample
{
	static void Add(string str1, string str2)
	{
		Console.WriteLine("Adding Two Strings: " + str1 + str2);
	}

	static void Add(int num1, int num2)
	{
		Console.WriteLine("Adding Two Integers: " + (num1 + num2));
	}

	static void Main(string[] args)
	{
		Add("John ", "Smith");
		Add(5, 10);
	}
}
```

Output:

```
Adding Two Strings: John Smith
Adding Two Integers: 15
```

Method Overloading

- Method overloading allows a class to have multiple methods with the same name but with a different signature.
- Methods can be overloaded based on the:
  - Number of parameters
  - Type of parameters (`int`, `double` etc.)
  - Kind of parameters (value, `ref`, `out`)

- **Note:** Methods cannot be overloaded based on return type or `params` keyword.

Method Overloading Example

- Can overload based on *number of parameters*:

```
public void Add(int n1, int n2)
{
      // method body
}

public void Add(int n1, int n2, int n3)
{
      // method body
}
```

Method Overloading Example

- Can overload based on *type of parameters*:

```
public void Add(int n1, int n2)
{
      // method body
}

public void Add(string n1, string n2)
{
      // method body
}
```

Method Overloading Example

- Can overload based on *kind of parameters*:

```
public void Add(int n1, int n2, int n3)
{
      // method body
}

public void Add(int n1, int n2, ref int n3)
{
      // method body
}
```

Method Overloading Example

- Can not overload based on *return type*:

```
public void Add(int n1, int n2)
{
      // method body
}

public int Add(int n1, int n2)
{
      // method body
}
```

- Will result in compile-time error.

Method Overloading Example

- Can not overload based on *params keyword*:

```
static void PrintArray(int[] array)
{
      // method body
}

static void PrintArray(params int[] array)
{
      // method body
}
```

- Will result in compile-time error.

Dynamic or Runtime Polymorphism

- In dynamic polymorphism, the *decision is made at run-time*.
- Run-time polymorphism is achieved by *method overriding*.
- Method overriding allows us to have methods in the base and derived classes with the same name and the same parameters.
- *By runtime polymorphism, we can point to any derived class from the object of the base class at runtime.*
- Through the reference variable of a base class, the determination of the method to be called is based on the object being referred to by reference variable.
- Compiler would not be aware whether the method is available for overriding the functionality or not.
- So compiler would not give any error at compile time.
- At runtime, it will be decided which method to call and if there is no method at runtime, it will give an error.

Example

```
public class BaseClass
{
      public virtual void Show()
      {
            Console.WriteLine("Show From Base Class");
      }
}
public class DerivedClass : BaseClass
{
      public override void Show()
      {
            Console.WriteLine("Show From Derived Class");
      }
}
static void Main(string[] args)
{
      BaseClass objBase;
      objBase = new BaseClass();
      objBase.Show();          // Output ----> Show From Base Class
      objBase = new DerivedClass();
      objBase.Show();          // Output ----> Show From Derived Class
}
```

Output:

```
Show From Base Class
Show From Derived Class
```

Virtual Method

- According to Microsoft documentation, “The `virtual` keyword is used to modify a method, property, indexer, or event declaration and allow for it to be overridden in a derived class”.
  - https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/virtual
- Virtual method is a method whose behavior can be overridden in derived class.
- Virtual method allows us to declare a method in base class that can be redefined in each derived class.
- When a virtual method is invoked, the run-time type of the object is checked for an overriding member.
- By default, methods are non-virtual. You cannot override a non-virtual method.
- You cannot use the `virtual` modifier with the `static`, `abstract`, `private` or `override` modifiers.

Explicit Polymorphism

- In C#, polymorphism is explicit - you must have a `virtual` modifier on the base class method and an `override` on the derived class method.
- If you don't put a modifier on a base class method, polymorphism can't happen.
- If you then add a non-overridden method to the derived class with the same signature as the non-virtual base class method, the compiler will generate a warning message.

Example

```
public class BaseClass
{
        public void Show()       // No virtual keyword
      {
            Console.WriteLine("Show From Base Class");
      }
}
public class DerivedClass : BaseClass
{
      // 'DerivedClass.Show()' hides inherited member 'BaseClass.Show()'.
      // Use the new keyword if hiding was intended.
      public void Show()        // No override keyword
      {
            Console.WriteLine("Show From Derived Class");
      }
}
static void Main(string[] args)
{
      BaseClass objBase;
      objBase = new BaseClass();
      objBase.Show();           // Output ----> Show From Base Class
      objBase = new DerivedClass();
      objBase.Show();           // Output ----> Show From Base Class
}
```

Output:

```
Show From Base Class
Show From Base Class
```

Implicit Polymorphism

- In other languages, Java for instance, you have what is called "implicit" polymorphism, where just putting the method in the derived class with the same signature as a base class method will enable polymorphism.
- In Java, all methods of a class are virtual by default unless the developer decides to use the `final` keyword, thus preventing subclasses from overriding that method.

- In contrast, C# adopts the strategy used by C++ where the developer must use the `virtual` keyword for subclasses to override the method.
- Thus all methods in C# are non-virtual by default.

Difference between Method Overriding and Method Hiding

- Method overriding allows a subclass to provide a specific implementation of a method that is already provided by base class.
- The implementation in the subclass overrides (replaces) the implementation in the base class.
- The important thing to remember about overriding is that the method that is doing the overriding is related to the method in the base class.

- When a virtual method is called, the actual type of the object to which the reference refers is used to determine which method implementation should be used.
- When a method of a base class is overridden in a derived class, the version defined in the derived class is used.

Difference between Method Overriding and Method Hiding

- Method hiding does not have a relationship between the methods in the base class and derived class.
- The method in the base class hides the method in the derived class.

Example

```
public class BaseClass
{
        public void Show()
      {
            Console.WriteLine("Show From Base Class");
      }
}
public class DerivedClass : BaseClass
{
      public new void Show()
      {
            Console.WriteLine("Show From Derived Class");
      }
}
static void Main(string[] args)
{
      BaseClass objBase;
      objBase = new BaseClass();
      objBase.Show();            // Output ----> Show From Base Class
      objBase = new DerivedClass();
      objBase.Show();            // Output ----> Show From Base Class
}
```

Output:

```
Show From Base Class
Show From Base Class
```

Abstract Classes and Methods

- When we think of a class type, we assume that apps will create objects of that type.
- In some cases, it's useful to declare classes for which you never intend to instantiate objects.
- Such classes are called *abstract classes*.
- These classes cannot be used to instantiate objects, because abstract classes are incomplete.
- Derived classes must define the "missing pieces".

Purpose of an Abstract Class

- The purpose of an abstract class is primarily to provide an appropriate base class from which other classes can inherit, and thus share a common design.
- Abstract classes are too general to create real objects.
- They specify only what is common among derived classes.

- Classes that can be used to instantiate objects are called *concrete classes*.
- Such classes provide implementations of every method they declare.
- Concrete classes provide the specifics needed to instantiate objects.

Creating an Abstract Class

- You make a class abstract by declaring it with the keyword `abstract`.
- An abstract class normally contains one or more abstract methods.
- An abstract method is one with keyword `abstract` in its declaration.

```
public abstract class BaseClass
{
    public abstract void MethodName(); // abstract method
}
```

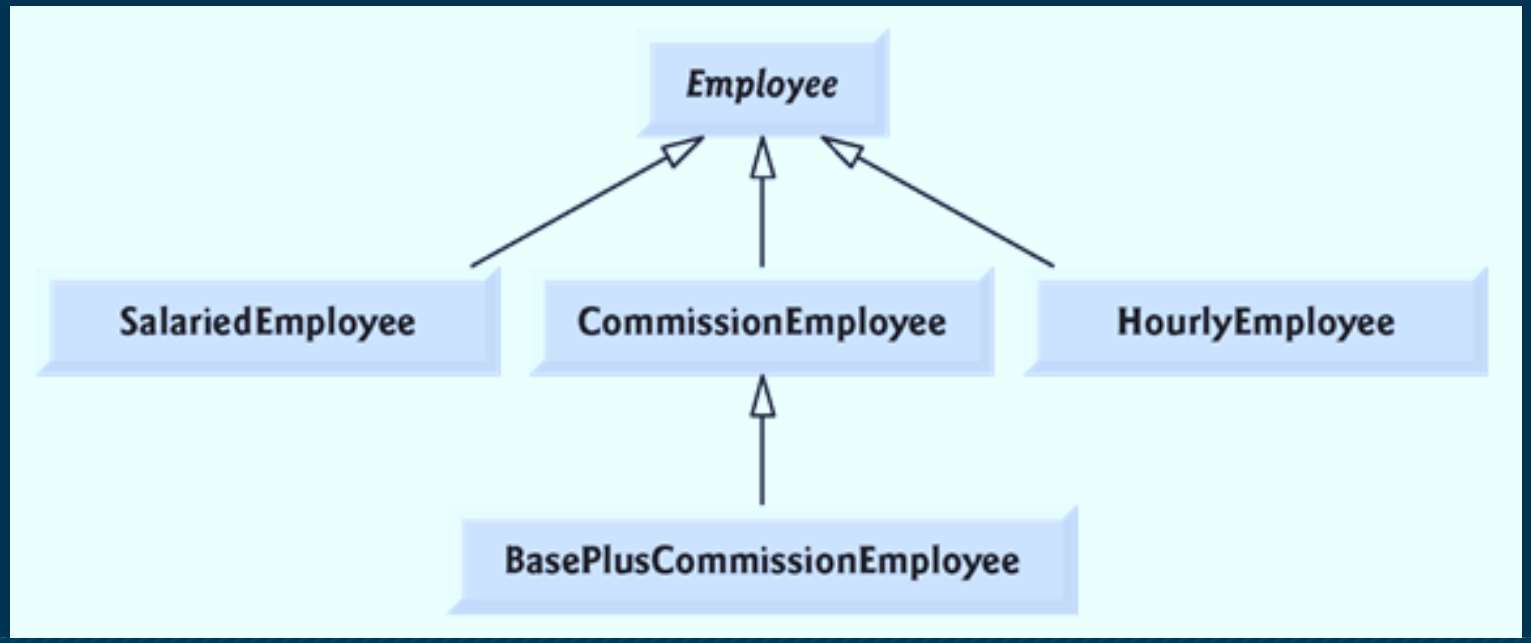
- Abstract methods are implicitly virtual and do not provide implementations.
- A class that contains abstract methods must be declared as an abstract class even if it contains some concrete (non-abstract) methods.
- Each concrete derived class of an abstract base class must provide implementations of the base class's abstract methods.

Exercise

- A company pays its employees on a weekly basis.
- The employees are of four types:
  - Salaried employees are paid a fixed weekly salary regardless of the number of hours worked.
  - Hourly employees are paid by the hour and receive "time-and-a-half" overtime pay for all hours worked in excess of 40 hours.
  - Commission employees are paid a percentage of their sales.
  - Salaried-commission employees receive a base salary plus a percentage of their sales.
- For the current pay period, the company has decided to reward salaried-commission employees by adding 10% to their base salaries.
- The company wants to implement an app that performs its payroll calculations polymorphically.

Exercise - *Hints*

- Use class `Employee` to represent the general concept of an employee.
- `SalariedEmployee`, `CommissionEmployee` and `HourlyEmployee` extend `Employee`.
- Class `SalaryPlusCommissionEmployee` – which extends `CommissionEmployee` – represents the last employee type.



Exercise - *Hints*

- Base class `Employee` provides method `Earnings`, in addition to the properties that manipulate `Employee`'s instance variables.
- Each earnings calculation depends on the employee's class, so we override `Earnings` in each class.
- The app iterates through the array and calls method `Earnings` for each `Employee` object. These method calls are processed polymorphically.

Exercise - *Hints*

- The diagram shows each of the five classes in the hierarchy down the left side and methods `Earnings` and `ToString` across the top.
- For each class, the diagram shows the desired results of each method.

	Earnings	ToString
Employee	abstract	*firstName lastName* social security number: *SSN*
Salaried-Employee	weeklySalary	salaried employee: *firstName lastName* social security number: *SSN* weekly salary: *weeklysalary*
Hourly-Employee	*If hours <= 40* wage * hours *If hours > 40* 40 * wage + (hours - 40) * wage * 1.5	hourly employee: *firstName lastName* social security number: *SSN* hourly wage: *wage* hours worked: *hours*
Commission-Employee	commissionRate * grossSales	commission employee: *firstName lastName* social security number: *SSN* gross sales: *grossSales* commission rate: *commissionRate*
BasePlus-Commission-Employee	baseSalary + (commissionRate * grossSales)	base salaried commission employee: *firstName lastName* social security number: *SSN* gross sales: *grossSales* commission rate: *commissionRate* base salary: *baseSalary*

Do It Yourself!

- **Exercise: Payroll System Modification:**

- Modify the payroll system example to include private instance variable `_birthDate` in class `Employee`.
- Use type `DateTime` to represent an employee's birthday.
- Assume that payroll is processed once per month.
- Create an array of `Employee` variables to store references to the various employee objects.
- In a loop, calculate the payroll for each `Employee` (polymorphically), and add a $100.00 bonus to the person's payroll amount if the current month is the month in which the `Employee`'s birthday occurs.

Do It Yourself!

- Create a class `Shape` that has two methods `Area()` and `Perimeter()`, having return types as `double`.
- They don't need any implementation, therefore, make them `abstract`.
- Create a class `Circle` that inherits from `Shape`. The constructor initializes the `Radius` property. Override and implement the `Area()` and `Perimeter()` methods.
- Create another class `Square` that also inherits `Shape`. The constructor initializes the `Side` property. Override and implement the `Area()` and `Perimeter()` methods.
- In the `Main()` method, create a variable of `Shape`, and instantiate it with `Circle` and `Square` objects, respectively.
- Print the area and perimeter of both shapes.

Thank You

Copyright

The materials provided in class and in SLATE are protected by copyright. They are intended for the personal, educational uses of students in this course and should not be shared externally or on websites such as Chegg, Course Hero or OneClass. Unauthorized distribution may result in copyright infringement and violation of Sheridan policies.

Do It Yourself!

Material has been taken from:

- Visual C# 2012: How to Program:
- https://learning.oreilly.com/library/view/visual-c-2012/9780133380170/ch11.html

- Professional C# 7 and .NET Core 2.0:
- https://learning.oreilly.com/library/view/professional-c-7/9781119449270/c04.xhtml