# Session 6: Inheritance and Polymorphism

Objectives

- **Define and describe inheritance**
- **Explain method overriding**
- **Define and describe sealed classes**
- **Explain polymorphism**

Definition of Inheritance 1-2

◆ The similarity in physical features of a child to that of its parents is due to the child having inherited features from its parents.

◆ Similarly, in C#, inheritance allows you to create a class by deriving the common attributes and methods of an existing class.

◆ The process of creating a new class by extending some features of an existing class is known as inheritance.

Inheritance in Real World
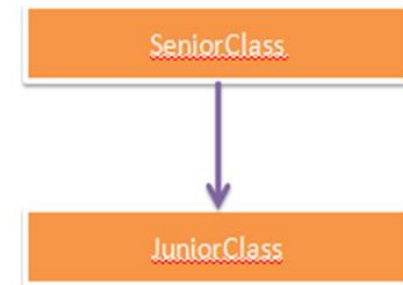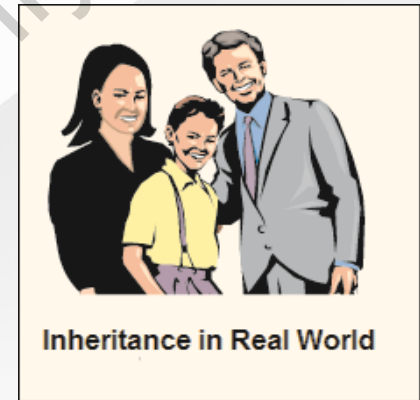
Definition of Inheritance 2-2

◆ Consider a class called **Vehicle** that consists of a variable called **color** and a method called **Speed()**.

◆ These data members of the **Vehicle** class can be inherited by the **TwoWheelerVehicle** and **FourWheelerVehicle** classes.

◆ Following figure illustrates an example of inheritance:
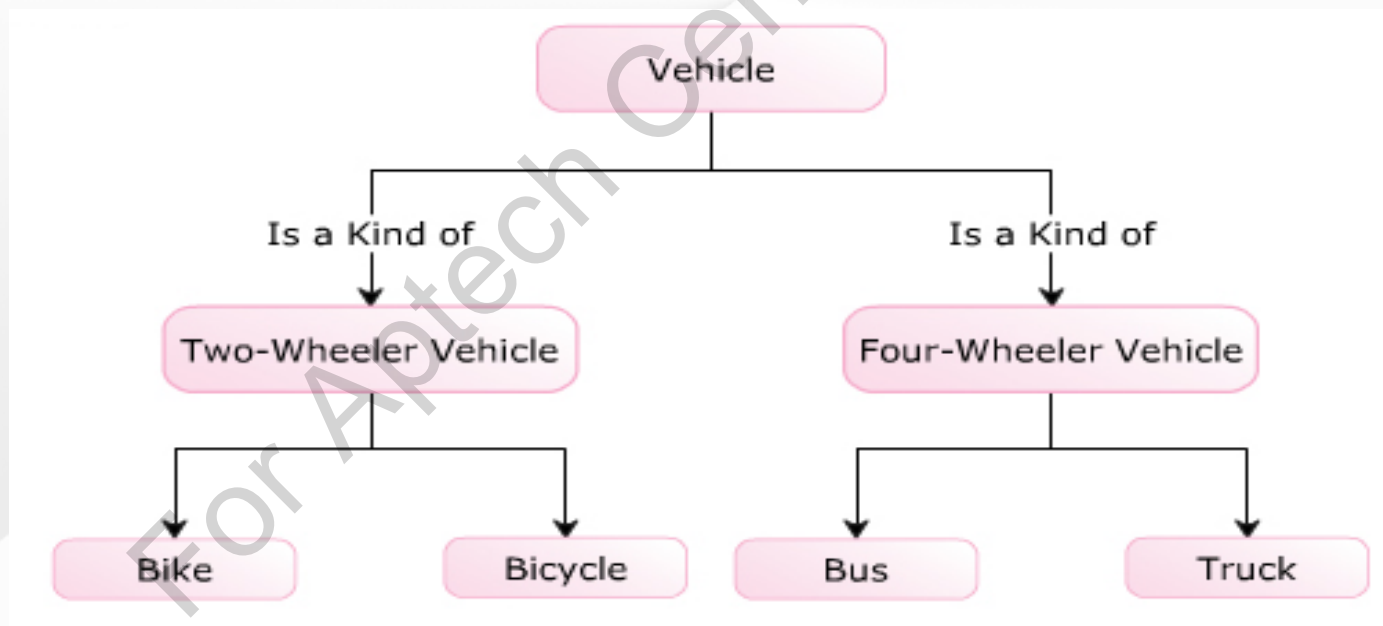
Purpose 1-3

◆ Reusability of a code enables you to use the same code in different applications with little or no changes.

Example

◆ Consider a class named **Animal** which defines attributes and behavior for animals.

◆ If a new class named **Cat** has to be created, it can be done based on **Animal** because a cat is also an animal.
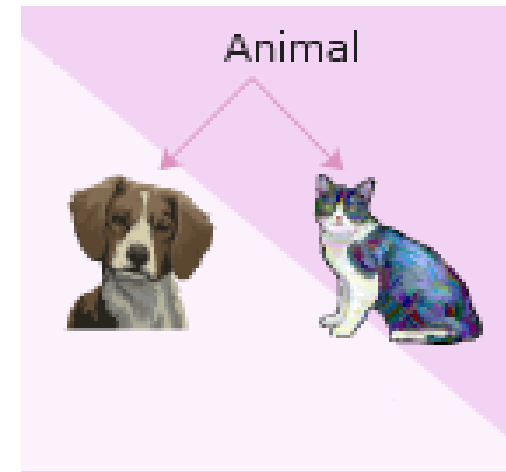
Purpose 2-3

◆ Apart from reusability, inheritance is widely used for:

Generalization
- Inheritance allows you to implement generalization by creating base classes.
- For example, consider the class Vehicle, which is the base class for its derived classes **Truck** and Bike.

Specialization
- Inheritance allows you to implement specialization by creating derived classes.
- For example, the derived classes such as Bike, Bicycle, Bus, and Truck are specialized by implementing only specific methods from its generalized base class Vehicle.

Extension
- Inheritance allows you to extend the functionalities of a derived class by creating more methods and attributes that are not present in the base class.
- It allows you to provide additional features to the existing derived class without modifying the existing code.

Purpose 3-3

◆ Following figure displays a real-world example demonstrating the purpose of inheritance:
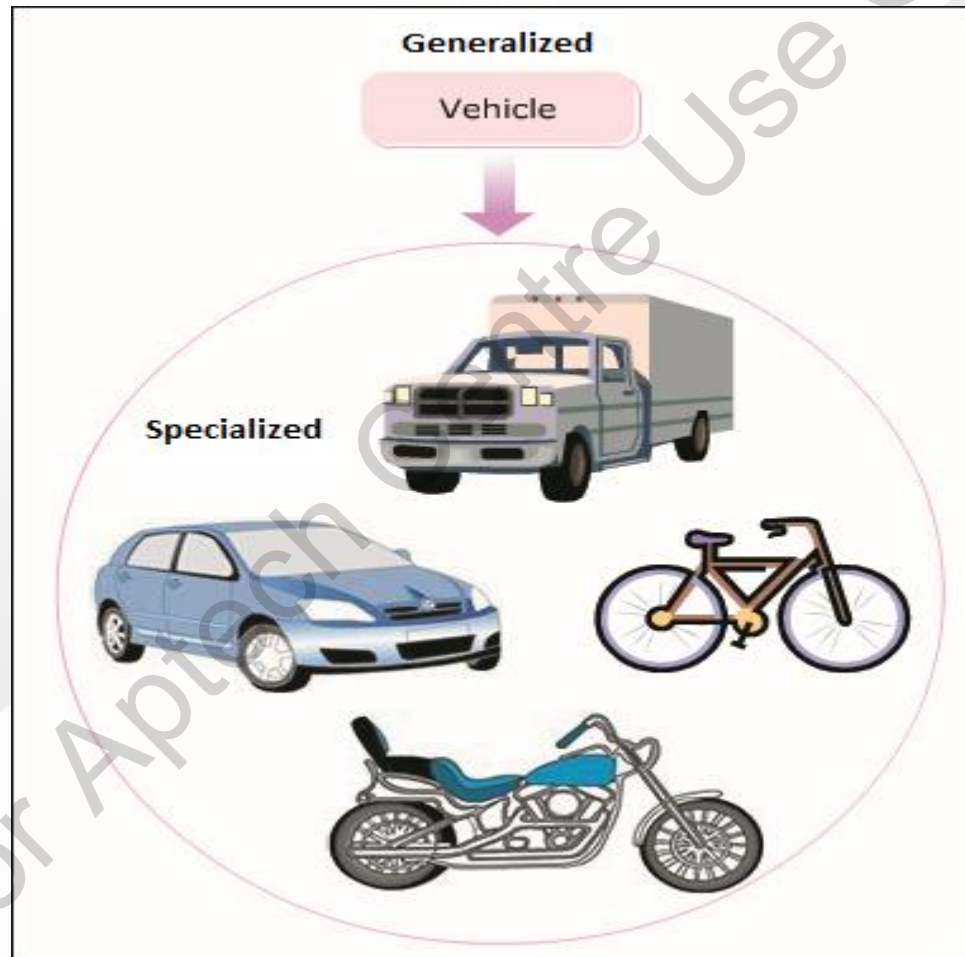
Multi-level Hierarchy

◆ Inheritance allows the programmer to build hierarchies that can contain multiple levels of inheritance.

> Example

◆ Consider three classes **Mammal**, **Animal,** and **Dog**. The class **Mammal** is inherited from the base class **Animal**, which inherits all the attributes of the **Animal** class.

◆ The class **Dog** is inherited from the class Mammal and inherits all the attributes of both the **Animal** and **Mammal** classes.

◆ Following figure depicts multi-level hierarchy of related classes:
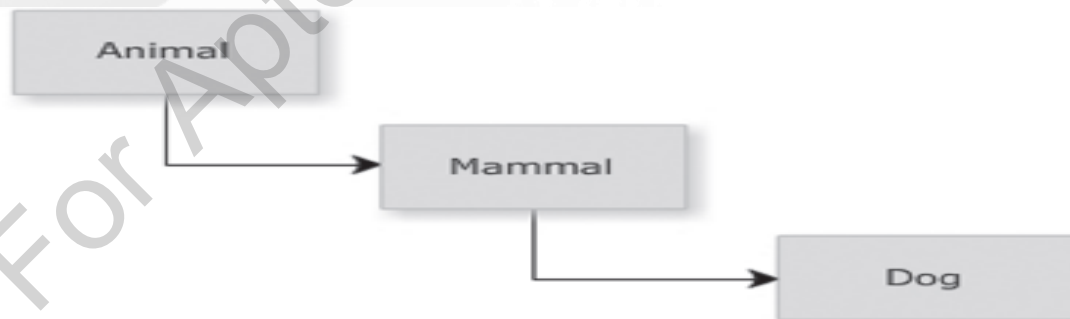
Implementing Inheritance 1-3

◆ To derive a class from another class in C#, insert a colon after the name of the derived class followed by the name of the base class.

◆ It can inherit all non-private methods and attributes of the base class.

◆ Following syntax is used to inherit a class in C#:

Syntax

```
<DerivedClassName>:<BaseClassName>
```

where,

◈ `DerivedClassName`: Is the name of the newly created child class.

◈ `BaseClassName`: Is the name of the parent class from which the current class is inherited.

Implementing Inheritance 2-3

◆ Following syntax is used to invoke a method of the base class:

Syntax	`<objectName>.<MethodName>;`

where,

◈ `objectName`: Is the object of the base class.

◈ `MethodName`: Is the name of the method of the base class.

◆ Following code demonstrates how to derive a class from another existing class and inherit methods from the base class:

Snippet

```
class Animal
{
    public void Eat()
    {
    Console.WriteLine("Every animal eats something.");
    }
    public void DoSomething()
    {
    Console.WriteLine("Every animal does something.");
    }
}
class Cat: Animal
{
    static void Main(string[] args)
    {
    Cat objCat = new Cat();
    objCat.Eat();
    objCat.DoSomething();
    }
}
```

Implementing Inheritance 3-3

◆ In the code:

◈ The class **Animal** consists of two methods, **Eat()** and **DoSomething()**. The class **Cat** is inherited from the class **Animal**.

◈ The instance of the class **Cat** is created and it invokes the two methods defined in the class **Animal**.

◈ When the instance of the class **Cat** invokes the **Eat()** and **DoSomething()** methods, the statements in the **Eat()** and **DoSomething()** methods of the base class **Animal** are executed.

Output

```
Every animal eats something.
Every animal does something.
```

`protected` Access Modifier 1-3

- The `protected` access modifier protects the data members that are declared using this modifier.

- The `protected` access modifier is specified using the `protected` keyword.

- Following figure displays an example of using the protected access modifier:
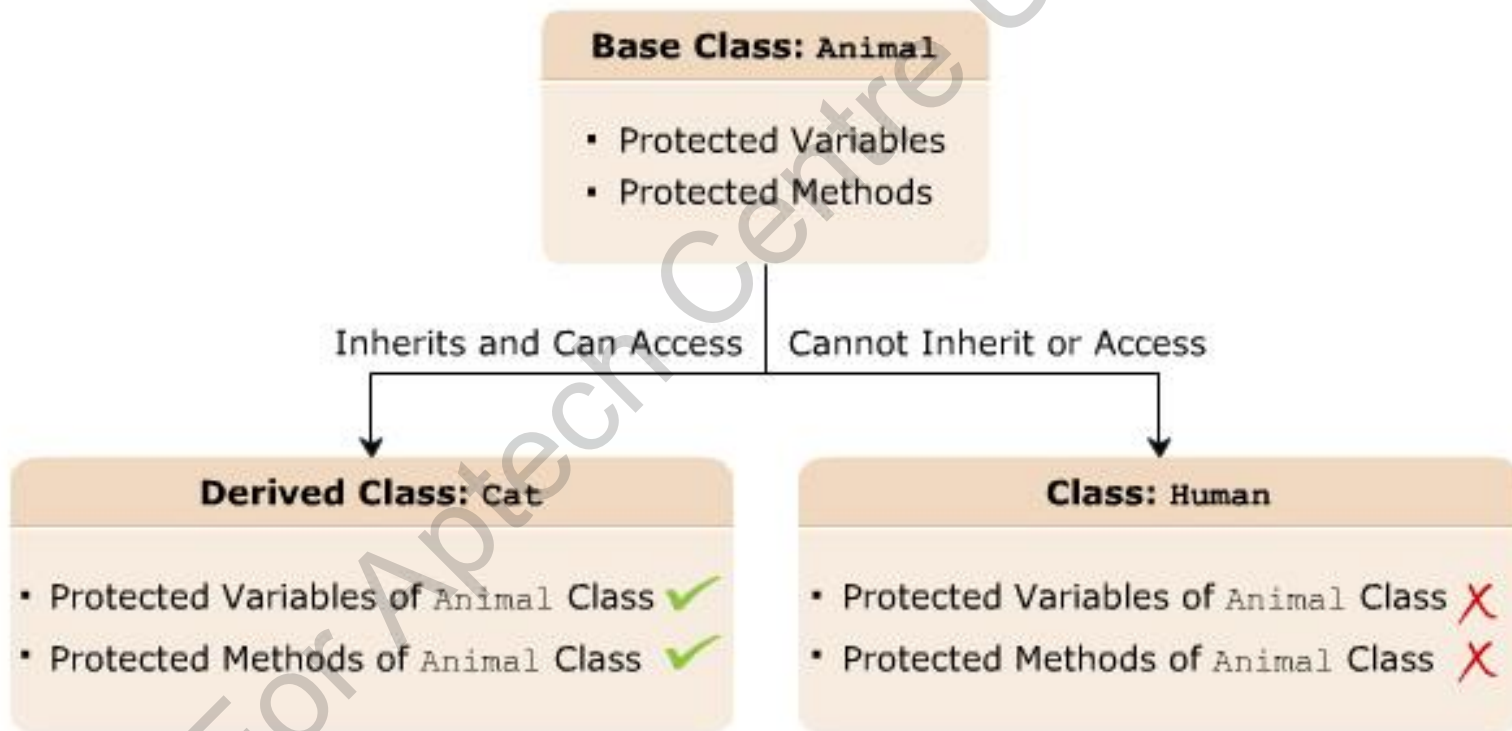
`protected` Access Modifier 2-3

◆ Following syntax declares a protected variable:

Syntax
`protected <data_type> <VariableName>;`

where,

◈ `data_type`: Is the data type of the data member.

◈ `VariableName`: Is the name of the variable.

`protected` Access Modifier 3-3

◆ Following code demonstrates the use of the `protected` access modifier:

```
class Animal {
        protected string Food;
        protected string Activity;
}
class Cat: Animal {
    static void Main(string[] args)
    {
     cat objCat = new Cat();
     objCat.Food="Mouse";
     objCat.Activity="laze around";
     Console.WriteLine("The Cat loves to eat"+objCat.Food+".");
     Console.WriteLine("The Cat loves to "+objCat.Activity+".");
}
}
```

◆ In the code:

   ◈ Two variables are created in the class **Animal** with the protected keyword.

   ◈ The class **Cat** is inherited from the class **Animal.**

Output

```
The Cat loves to eat Mouse.
The Cat loves to laze around.
```

base Keyword

◆ The base keyword allows you to do the following:

Access the variables and methods of the base class from the derived class.	Re-declare the methods and variables defined in the base class.	Invoke the derived class data members.	Access the base class members using the base keyword.

new Keyword

◆ The `new` keyword can either be used as an operator or as a modifier in C#.

new Operator — Instantiates a class by creating its object which invokes the constructor of the class.

Modifier — Hides the methods or variables of the base class that are inherited in the derived class.

◆ This allows you to redefine the inherited methods or variables in the derived class.

◆ Since redefining the base class members in the derived class results in base class members being hidden.

Using new Keyword 1-2

Following code demonstrates the use of the new modifier to redefine the inherited methods in the base class:

Snippet

```
class Employees
{
    int _empId=1;
    string _empName="James Anderson";
    int _age=25;
    public void Display()
    {
        Console.WriteLine("Employee ID:"+_empId);
        Console.WriteLine("Employee Name:"+_empName);
    }
}
class Department: Employees
{
    int _deptId=501;
    string _deptName="Sales";
    new void Display()
    {
        base.Display();
        Console.WriteLine("Department ID:"+_deptId);
        Console.WriteLine("Department Name:"+_deptName);
    }
    static void Main(string[] args)
    {
        Department objDepartment = new Department();
        objDepartment.Display();
    }
}
```

Using new Keyword 2-2

◆ In the code:

  ◈ The class **Employees** declares a method called **Display()**.

  ◈ This method is inherited in the derived class **Department** and is preceded by the new keyword.

  ◈ The new keyword hides the inherited method **Display()** that was defined in the base class, thereby executing the **Display()** method of the derived class when a call is made to it.

Output

```
Employee ID: 1
Employee Name: James Anderson
Department ID: 501
Department Name: Sales
```

Constructor Inheritance

- In C#, you can:

Invoke the base class constructor by either instantiating the derived class or the base class.

Invoke the constructor of the base class followed by the constructor of the derived class.

Invoke the base class constructor by using the base keyword in the derived class constructor declaration.

Pass parameters to the constructor.

- However, C# cannot inherit constructors similar to how you inherit methods.

- Following figure displays an example of constructor inheritance:


```
Base Class: Animal
{
Animal()
{}
}
Derived Class: Cat
{
Cat()
{}
public void Create(){
Cat objCat = new Cat();
}
}
```
Invoked First →

Invoked Second
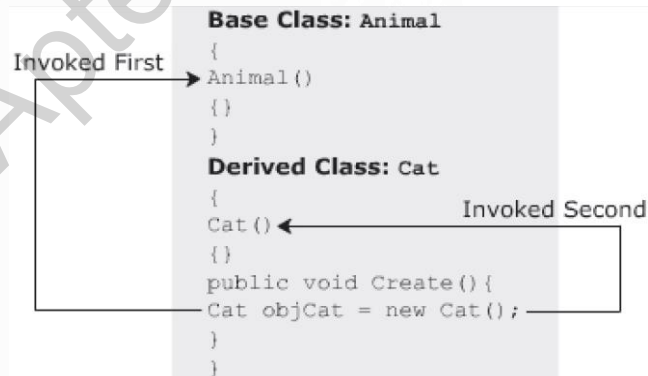
# Invoking Parameterized Base Class Constructors

◆ The derived class constructor can explicitly invoke the base class constructor by using the `base` keyword.

◆ If a base class constructor has a parameter, the `base` keyword is followed by the value of the type specified in the constructor declaration.

◆ If there are no parameters, the `base` keyword is followed by a pair of parentheses.

Method Overriding

- Method overriding:
  - Allows the derived class to override or redefine the methods of the base class.
  - Ensures reusability while inheriting classes.
  - Is implemented in the derived class from the base class is known as the Overridden Base Method.

- Following figure depicts method overriding:
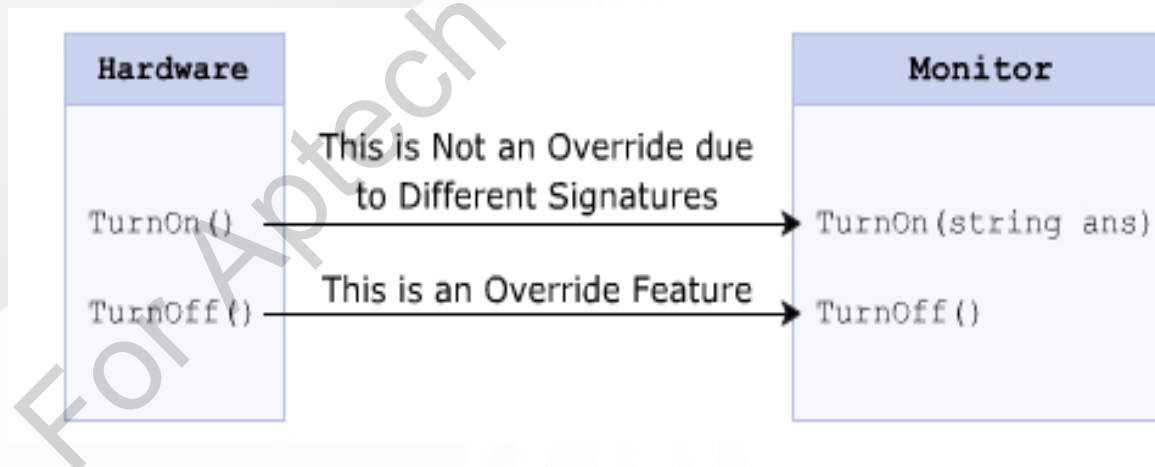
`virtual` and `override` Keywords 1-3

◆ You can override a base class method in the derived class using appropriate C# keywords such as:

To override a particular method of the base class in the derived class, you must declare the method in the base class using the virtual keyword.

A method declared using the virtual keyword is referred to as a virtual method.

The `override` keyword overrides the base class method in the derived class.

`virtual` and `override` Keywords 2-3

◆ Following is the syntax for declaring a virtual method using the `virtual` keyword:

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

where,

◈ `access_modifier`: Is the access modifier of the method, which can be `private`, `public`, `protected`, or `internal`.

◈ `virtual`: Is a keyword used to declare a method in the base class that can be overridden by the derived class.

◈ `return_type`: Is the type of value the method will return.

◈ `MethodName`: Is the name of the virtual method.

◈ `parameter-list`: Is the parameter list of the method; it is optional.

`virtual` and `override` Keywords 3-3

◆ Following is the syntax for overriding a method using the `override` keyword:

```
<accessmodifier> override <returntype> <MethodName>
(<parameters-list>)
```

where,

◈ `override`: Is the keyword used to override a method in the derived class.

Calling the Base Class Method 1-3

◆ Method overriding allows the derived class to redefine the methods of the base class.

◆ It allow the base class methods to access the new method but not the original base class method.

◆ To execute the base class method as well as the derived class method, you can create an instance of the base class.

Calling the Base Class Method 2-3

◆ Following code demonstrates how to access a base class method:

Snippet

```csharp
class Student {
string _studentName = "James";
string _address = "California";
public virtual void PrintDetails() {

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

Calling the Base Class Method 3-3

◆ In the code:

  ◈ The class **Student** consists of a virtual method called **PrintDetails()**.

  ◈ The class **Grade** inherits the class **Student** and overrides the base class method **PrintDetails()**.

  ◈ The Main() method creates an instance of the base class **Student** and the derived class **Grade**.

Output

```
Student Name: James
Address: California
Class: Four
Percentage: 71.25
```

Sealed Classes 1-3

◆ Features of a sealed class are as follows:

> Can be declared by preceding the class keyword with the sealed keyword.

> Prevents a class from being inherited by any other class.

:

> Cannot be a base class as it cannot be inherited by any other class.

Syntax

```
sealed class<ClassName>
{
//body of the class
}
```

where,

◈ `sealed`: Is a keyword used to prevent a class from being inherited.

◈ `ClassName`: Is the name of the class that must be sealed.

Sealed Classes 2-3

◆ Following code demonstrates the use of a sealed class in C# which will generate a compiler error:

Snippet

```
sealed class Product
{
   public int Quantity;
   public int Cost;
}
class Goods
{
    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);
    }
}
class Pen : Product
{
}
```

◆ In the code:

◈ The class **Product** is declared as sealed and it consists of two variables.

◈ The class **Goods** contains the code to create an instance of **Product** and uses the dot (.) operator to invoke variables declared in **Product**.

Sealed Classes 3-3

◆ The class **Pen** tries to inherit the sealed class **Product**, the C# compiler generates an error, as shown in the following figure:

Error List

🔽 ▾	❌ 1 Error	⚠ 0 Warnings	ⓘ 0 Messages				Search Error List
Description ▲			File	Line	Column	Project
❌ 1	'ConsoleApplication1.Pen': cannot derive from sealed type 'ConsoleApplication1.Product'			Goods.cs	26	11	ConsoleApplication1

Purpose of Sealed Classes

◆ Consider a class named `SystemInformation` that consists of critical methods that affect the working of the operating system.

◆ You might not want any third party to inherit the class `SystemInformation` and override its methods, thus, causing security and copyright issues.

◆ Here, you can declare the `SystemInformation` class as sealed to prevent any change in its variables and methods.

Guidelines

◆ Sealed classes are restricted classes that cannot be inherited where the list depicts the conditions in which a class can be marked as sealed:

⬥ If overriding the methods of a class might result in unexpected functioning of the class.

⬥ When you want to prevent any third party from modifying your class.

Sealed Methods 1-3

In C#, a method cannot be declared as sealed.

Sealing the new method prevents the method from further overriding.

An overridden method can be sealed by preceding the override keyword with the `sealed` keyword.

```
sealed override <return_type> <MethodName>{}
```

where,

◈ `return_type`: Specifies the data type of value returned by the method.

◈ `MethodName`: Specifies the name of the overridden method.

Sealed Methods 2-3

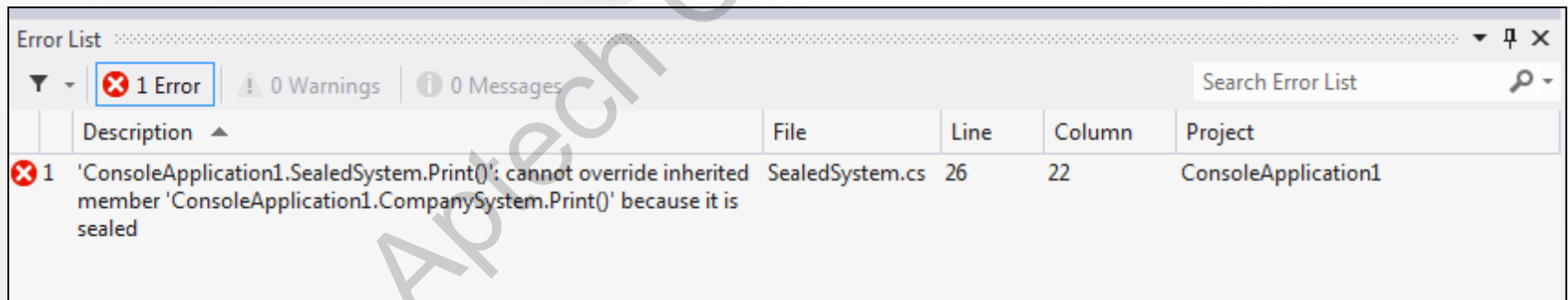- Following code declares an overridden method `Print()` as sealed:

Snippet

```
using System;
class ITSystem {
public virtual void Print() {
Console.WriteLine ("The system should be handled carefully");
    }
}
class CompanySystem : ITSystem {
public override sealed void Print()
{
  Console.WriteLine ("The system information is confidential");
  Console.WriteLine ("This information should not be overridden");
    }
}
class SealedSystem : CompanySystem {
public override void Print() {
 Console.WriteLine ("This statement won't get executed");
}
static void Main (string [] args) {
 SealedSystem objSealed = new SealedSystem();
 objSealed.Print();
}
}
```

Sealed Methods 3-3

- In the code:

  - The class **ITSystem** consists of a virtual function **Print()**.

  - The class **CompanySystem** is inherited from the class **ITSystem**.

  - The overridden method **Print()** is sealed by using the `sealed` keyword, which prevents further overriding of that method.

  - When the class **SealedSystem** overrides the sealed method **Print()**, the C# compiler generates an error as shown in the following figure:

Error List				
▼ ▼ ⊗ 1 Error ⚠ 0 Warnings ⓘ 0 Messages			Search Error List	🔎 ▼
Description ▲	File	Line	Column	Project
⊗ 1 'ConsoleApplication1.SealedSystem.Print()': cannot override inherited member 'ConsoleApplication1.CompanySystem.Print()' because it is sealed	SealedSystem.cs	26	22	ConsoleApplication1

Polymorphism

◆ Polymorphism is derived from two Greek words, namely **Poly** and **Morphos**, meaning forms.

◆ Following methods in a class have the same name, but different signatures that perform the same basic operation, but in different ways:

  ◈ `Area(float radius)`

  ◈ `Area(float base, float height)`

◆ These methods calculate the area of the circle and triangle taking different parameters and using different formulae.

Example

```
Shape

SetDimensions(int length,
              int breadth)         Overloaded Methods –
SetDimensions(int length,          Way of Implementing
              int breadth,         Polymorphism
              int height)
```

Implementation 1-2

◆ Polymorphism can be implemented in C# through method overloading and method overriding.

You can create multiple methods with the same name in a class or in different classes having different method body or different signatures.

Methods having the same name, but different signatures in a class are referred to as overloaded methods.

Methods inherited from the base class in the derived class and modified within the derived class are referred to as overridden methods.

Implementation 2-2

◆ Following code demonstrates the use of method overloading feature:

Snippet

```
class Area {
static int CalculateArea(int len, int wide) {
return len * wide;
}
static double CalculateArea(double valOne, double valTwo) {
        return 0.5 * valOne * valTwo;
}
static void Main(string[] args) {
  int length = 10;
  int breadth = 22;
  double tbase = 2.5;
  double theight = 1.5;
  Console.WriteLine("Area of Rectangle: " + CalculateArea(length, breadth));
  Console.WriteLine("Area of triangle: " + CalculateArea(tbase, theight));
  }
}
```

Output

```
Area of Rectangle: 220
Area of triangle: 1.875
```

Compile-time and Run-time Polymorphism 1-3

- Polymorphism can be broadly classified into the following categories:
    - Compile-time polymorphism
    - Run-time polymorphism
- Following table differentiates between compile-time and run-time polymorphism:

Compile-time Polymorphism	Run-time Polymorphism
Is implemented through method **overloading**.	Is implemented through method **overriding**.
Is executed at the compile-time since the compiler knows which method to execute depending on the number of parameters and their data types.	Is executed at run-time since the compiler does not know the method to be executed, whether it is the base class method that will be called or the derived class method.
Is referred to as **static** polymorphism.	Is referred to as **dynamic** polymorphism.

Compile-time and Run-time Polymorphism 2-3

◆ Following code demonstrates the implementation of run-time polymorphism:

Snippet

```
using System;
class Circle {
protected const double PI = 3.14;
protected double Radius = 14.9;
public virtual double Area()
{
return PI * Radius * Radius;
}
}
class Cone : Circle {
protected double Side = 10.2;
public override double Area()
{
return PI * Radius * Side;
}
static void Main(string[] args)
{
Circle objRunOne = new Circle();
Console.WriteLine("Area is: " + objRunOne.Area());
Circle objRunTwo = new Cone();
Console.WriteLine("Area is: " + objRunTwo.Area());
}
}
```

- In the code:

  - The class **Circle** initializes protected variables and contains a virtual method **Area()**that returns the area of the circle.

  - The class Cone is derived from the class **Circle**, which overrides the method **Area()**.

  - The **Area()**method returns the area of the cone by considering the length of the cone, which is initialized to the value 10.2.

Output

Area is: 697.1114

Area is: 477.2172

Summary

- Inheritance allows you to create a new class from another class, thereby inheriting its common properties and methods.

- Inheritance can be implemented by writing the derived class name followed by a colon and the name of the base class.

- Method overriding is a process of redefining the base class methods in the derived class.

- Methods can be overridden by using a combination of `virtual` and `override` keywords within the base and derived classes respectively.

- Sealed classes are classes that cannot be inherited by other classes.

- You can declare a sealed class in C# by using the `sealed` keyword.

- Polymorphism is the ability of an entity to exist in two forms that are compile-time polymorphism and run-time polymorphism.