Microsoft® Official Course

Module 5

Creating a Class Hierarchy by Using Inheritance

Microsoft®

Module Overview

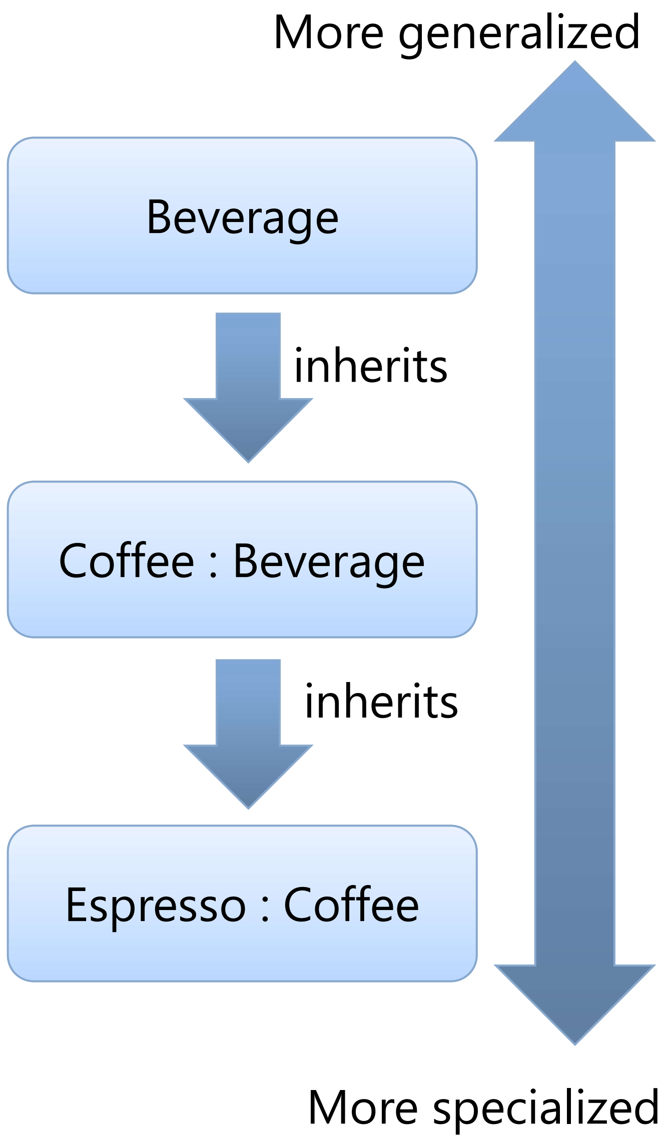
- Creating Class Hierarchies
- Extending .NET Framework Classes

Lesson 1: Creating Class Hierarchies

- What Is Inheritance?
- Creating Base Classes
- Creating Base Class Members
- Inheriting from a Base Class
- Calling Base Class Constructors and Members
- Demonstration: Calling Base Class Constructors

What Is Inheritance?

- inheritance enables you to create new classes (derived classes) by inheriting characteristics and behaviors from existing classes (base classes).
- The **derived class** is a more specialized instance of the **base class**.
  - The derived class inherits all the members of the base class, including constructors, methods, properties, fields, and events.
- Not only does this save you time by reducing the amount of code you need to write, it also enables you to create hierarchies of related classes that you can then use interchangeably, depending on your requirements.
- The concept of inheritance is one of the main pillars of object-oriented programming.



What Is Inheritance? - Example

- the terms derives and inherits are used interchangeably.
- though they are different … **inheriting from a class** and **implementing an interface** are both examples of inheritance.

```
public class Beverage
{
   protected int servingTemperature;
   public string Name { get; set; }
   public bool IsFairTrade { get; set; }
   public int GetServingTemperature()
   {
      return servingTemperature;
   }
}
```

```
public class Coffee : Beverage
{
   public string Bean { get; set; }
   public string Roast { get; set; }
   public string CountryOfOrigin { get; set; }
}
```

```
Coffee coffee1 = new Coffee();
// Use base class members.
coffee1.Name = "Fourth Espresso";
coffee1.IsFairTrade = true;
int servingTemp = coffee1.GetServingTemperature();
// Use derived class members.
coffee1.Bean = "Arabica";
coffee1.Roast = "Dark";
coffee1.CountryOfOrigin = "Columbia";
```

Creating Base Classes

- Use the **abstract** keyword to create a base class that cannot be instantiated

```
public abstract class Beverage
```

  - Use it when you want to create classes that serve solely as base classes for other types.
  - An abstract class can contain both abstract and non-abstract members.
    - You can only include abstract members within abstract classes. A non-abstract class cannot include abstract members
  - Any class that inherits from the abstract class must provide an implementation for the abstract members. Non-abstract members, however, can be used directly by derived classes.
  - The abstract class cannot be instantiated directly

```
abstract class Beverage
{
   // Non-abstract members.
   // Derived classes can use these members without modifying them.
   public string Name { get; set; }
   public bool IsFairTrade { get; set; }
   // Abstract members.
   // Derived classes must override and implement these members.
   public abstract int GetServingTemperature();
}
```

- Use the **sealed** keyword to create a class that cannot be inherited

```
public sealed class Tea : Beverage
```

  - Important note: a **sealed** class inherits from a base class.
  - Important note: the **sealed** keyword just prevents any **further** inheritance.
  - Any static class is also a sealed class. You can never inherit from a static class.
  - Any static members within non-static classes are not inherited by derived classes.

Creating Base Class Members

- Use the **virtual** keyword to create members that you can override in derived classes

```
public virtual int GetServingTemperature()
```

  - You can only override a base class member if the member is marked as **virtual** in the base class. You cannot override constructors

- When you create a class, you can use access modifiers to control whether the members of your class are accessible to derived types.

Access Modifier	Details
public	The member is available to code running in any assembly.
protected	The member is available only within the containing class or in classes derived from the containing class.
internal	The member is available only to code within the current assembly.
protected internal	The member is available to any code within the current assembly, and to types derived from the containing class in any assembly.
private	The member is available only within the containing class.

Creating Base Class Members – example

-
  ```
  public class Beverage
  {
    private int ServingTemperature1;
    protected int ServingTemperature2;
    internal int ServingTemperature3;
    protected internal int ServingTemperature4;
    public int ServingTemperature5;
  }
  ```

-
  ```
  public class Coffee : Beverage
  {
  }
  ```

  - ServingTemperature1 is not accessible within the Coffee class. You cannot use it as a private field within the Coffee class.
  - ServingTemperature2 is accessible within the Coffee class, but it is not accessible to consumers of the Coffee class. In other words, it is effectively a private field in the Coffee class.
  - ServingTemperature3 is accessible to any code within the current assembly. In other words, consumers of the Coffee class within the current assembly can read and write to this field.
  - ServingTemperature4 is accessible to any code within the current assembly, and is also available to derived classes in other assemblies. In other words, if the Coffee class is in a different assembly than the Beverage class, the ServingTemperature4 field is still accessible within the Coffee class.
  - ServingTemperature5 is universally accessible.

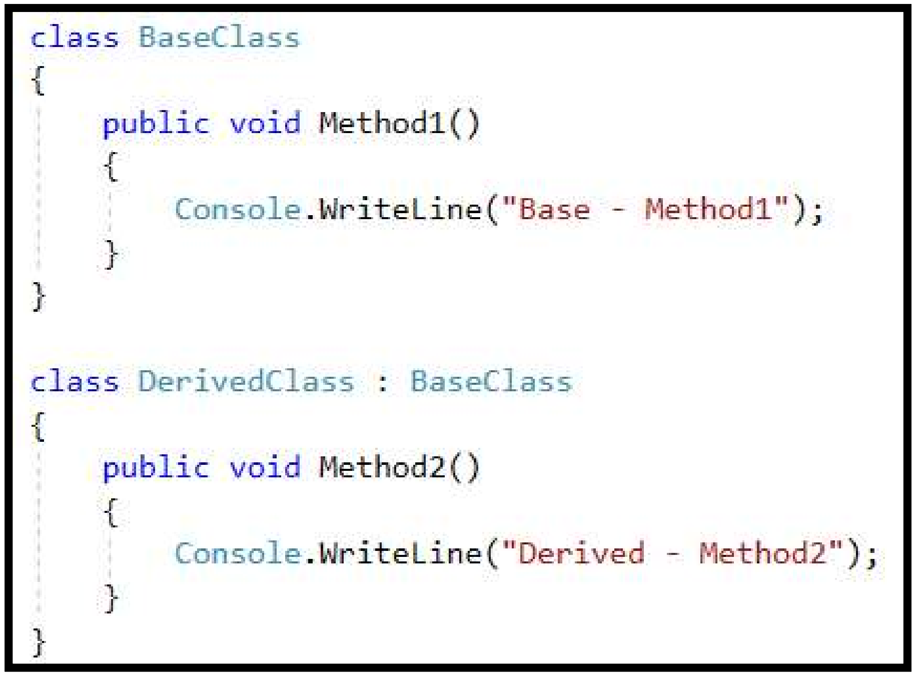
Inheriting from a Base Class

- To inherit from a base class, add the name of the base class to the class declaration

- The derived class inherits every member from the base class.
  - Within your derived class, you can add new members to extend the functionality of the base type.

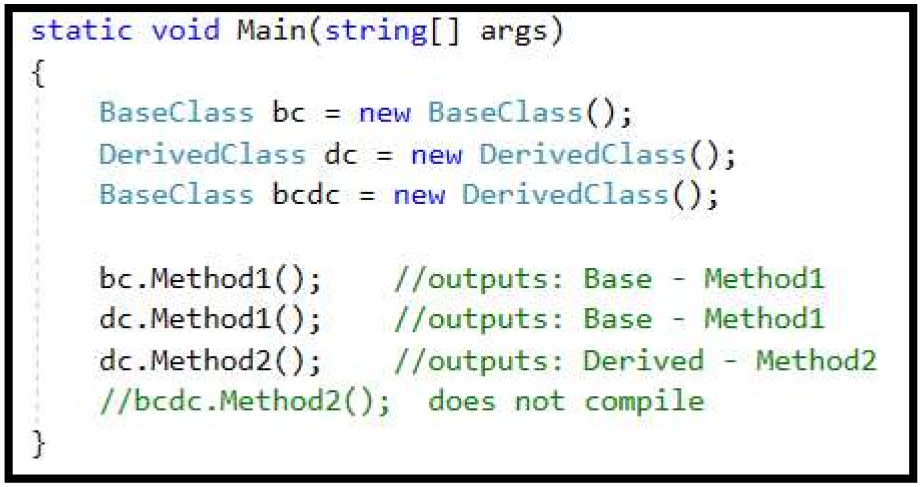
- A class can only inherit from one base class. But it can implement one or more interfaces in addition to deriving from a base type.

```
public class Coffee : Beverage
```



```
class BaseClass
{
    public void Method1()
    {
        Console.WriteLine("Base - Method1");
    }
}

class DerivedClass : BaseClass
{
    public void Method2()
    {
        Console.WriteLine("Derived - Method2");
    }
}
```



```
static void Main(string[] args)
{
    BaseClass bc = new BaseClass();
    DerivedClass dc = new DerivedClass();
    BaseClass bcdc = new DerivedClass();

    bc.Method1();     //outputs: Base - Method1
    dc.Method1();     //outputs: Base - Method1
    dc.Method2();     //outputs: Derived - Method2
    //bcdc.Method2();  does not compile
}
```

Inheriting from a Base Class

- To override virtual base class members, use the **override** keyword

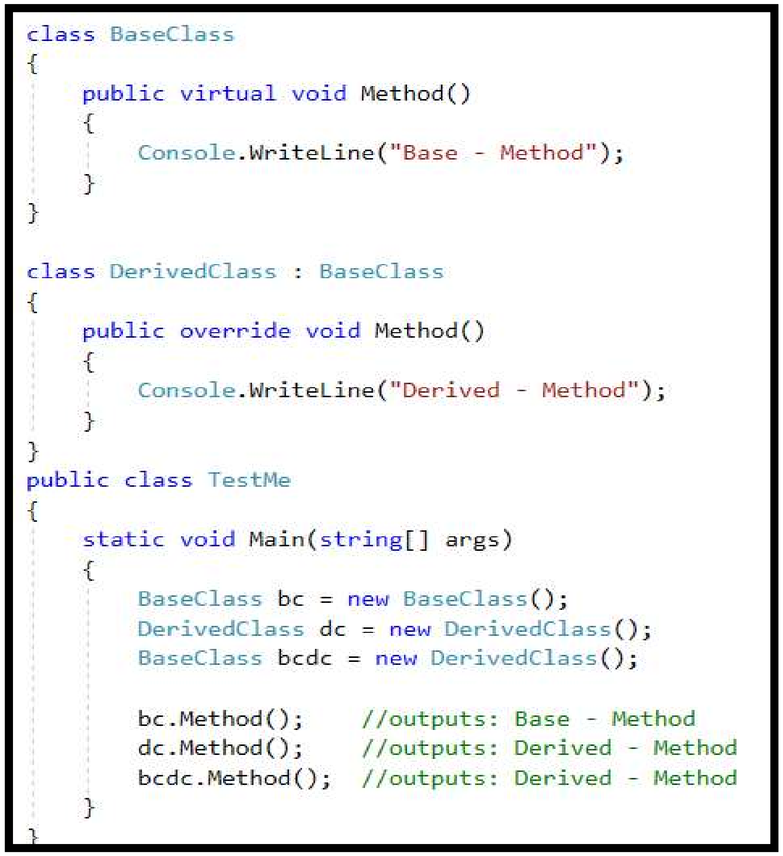
```
public override int GetServingTemperature()
```

- To prevent classes further down the class hierarchy from overriding your override methods, use the **sealed** keyword
  - can only apply the **sealed** modifier if the member is an **override** member
  - members are inherently sealed unless they are marked as **virtual**
    - if the base class method is marked as virtual, any descendants are able to override the method unless you seal it at some point in the class hierarchy.

```
sealed public override int GetServingTemperature()
```

Inheriting from a Base Class – example

- When you use the **override** keyword, your method **extends** the base class method.
  - Typically, that is the desired behavior in inheritance hierarchies. You want objects that have values that are created from the derived class to use the methods that are defined in the derived class. You achieve that behavior by using override to extend the base class method.

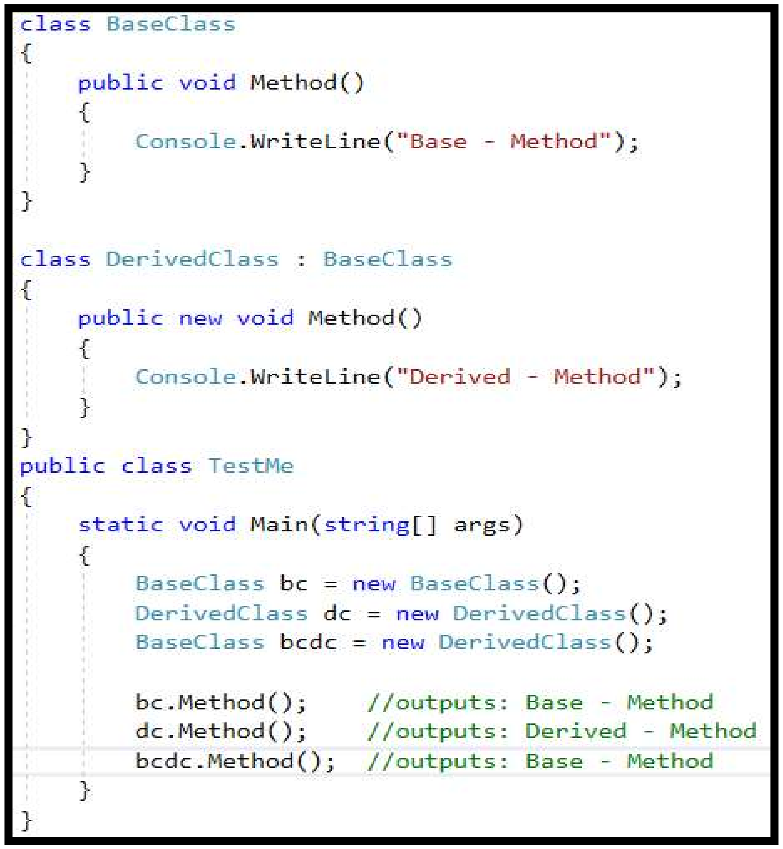


```
class BaseClass
{
    public virtual void Method()
    {
        Console.WriteLine("Base - Method");
    }
}

class DerivedClass : BaseClass
{
    public override void Method()
    {
        Console.WriteLine("Derived - Method");
    }
}
public class TestMe
{
    static void Main(string[] args)
    {
        BaseClass bc = new BaseClass();
        DerivedClass dc = new DerivedClass();
        BaseClass bcdc = new DerivedClass();

        bc.Method();    //outputs: Base - Method
        dc.Method();    //outputs: Derived - Method
        bcdc.Method();  //outputs: Derived - Method
    }
}
```

- When you use the **new** keyword, your method **hides** the base class method.
  - By using new, you are asserting that you are aware that the member that it modifies hides a member that is inherited from the base class
  - E.g. Animal, Cat:Animal …



```
class BaseClass
{
    public void Method()
    {
        Console.WriteLine("Base - Method");
    }
}

class DerivedClass : BaseClass
{
    public new void Method()
    {
        Console.WriteLine("Derived - Method");
    }
}
public class TestMe
{
    static void Main(string[] args)
    {
        BaseClass bc = new BaseClass();
        DerivedClass dc = new DerivedClass();
        BaseClass bcdc = new DerivedClass();

        bc.Method();    //outputs: Base - Method
        dc.Method();    //outputs: Derived - Method
        bcdc.Method();  //outputs: Base - Method
    }
}
```

- See also https://docs.microsoft.com/en-us/previous-versions/visualstudio/visual-studio-2012/ms173153(v=vs.110)

Calling Base Class Constructors and Members

- To call a base class constructor from a derived class, add the base constructor to your constructor declaration

```
public Coffee(string name, bool isFairTrade, int temp)
    : base(name, isFairTrade, servingTemp)
```

  - Pass parameter names to the base constructor as arguments
  - Do not use the base keyword within the constructor body
  - You cannot override constructors in derived classes.
    - Instead, when you create constructors in a derived class, your constructors will automatically call the default base class constructor before they execute any of the logic that you have added.

- To call base class methods from a derived class, use the base keyword like an instance variable

```
base.GetServingTemperature();
```

- The rules of inheritance do not apply to static classes & members. As such, you cannot use the base keyword within a static method.

Lesson 2: Extending .NET Framework Classes

- Inheriting from .NET Framework Classes
- Creating Custom Exceptions
- Throwing and Catching Custom Exceptions
- Inheriting from Generic Types
- Creating Extension Methods
- Demonstration: Refactoring Common Functionality into the User Class Lab

Inheriting from .NET Framework Classes

- Inherit from .NET Framework classes to:
  - Reduce development time
  - Standardize functionality
- Inherit from any .NET Framework type that is not **sealed** or **static**
- Override any base class members that are marked as **virtual**
- If you inherit from an **abstract** class, you must provide implementations for all **abstract** members.
- Choosing a base class wisely …:
  - If you find yourself replicating functionality that is available in built-in classes, you should probably **choose a more specific base class**.
  - If you find that you need to override several members, you should probably **choose a more general base class.**

Inheriting from .NET Framework Classes - Example

- For example, consider that you want to create a class that stores a linear list of values. The class must enable you to remove duplicate items from the list.
  - Rather than creating a new list class from nothing, you can accomplish this by creating a class that inherits from the generic **List<T>** class and adding a single method to remove duplicate items.

```
public class UniqueList<T> : List<T>
{
    public void RemoveDuplicates()
    {
        base.Sort();
        for (int i = this.Count - 1; i > 0; i--)
        {
            if(this[i].Equals(this[i-1]))
            {
                this.RemoveAt(i);
            }
        }
    }
}
```

Creating Custom Exceptions

- When you need to throw exceptions in your code, you should reuse existing .NET Framework exception types wherever possible. For example:
  - If you invoke a method with a null argument value, and the method cannot handle null argument values, the method will throw an **ArgumentNullException**.
  - If you attempt to divide a numerical value by zero, the runtime will throw a **DivideByZeroException**.
  - If you attempt to retrieve an indexed item from a collection, and the index it outside the bounds of the collection, the indexer will throw an **IndexOutOfRangeException**.

- When Should You Create a Custom Exception Type?
  - Existing exception types do not adequately represent the error condition you are identifying.
  - The exception requires very specific remedial action that differs from how you would handle built-in exception types.

- because all exceptions <u>ultimately</u> derive from **System.Exception**, a **catch** block that catches exceptions of type **Exception** will catch all exceptions.
  - This is why you should catch more specific exceptions first.

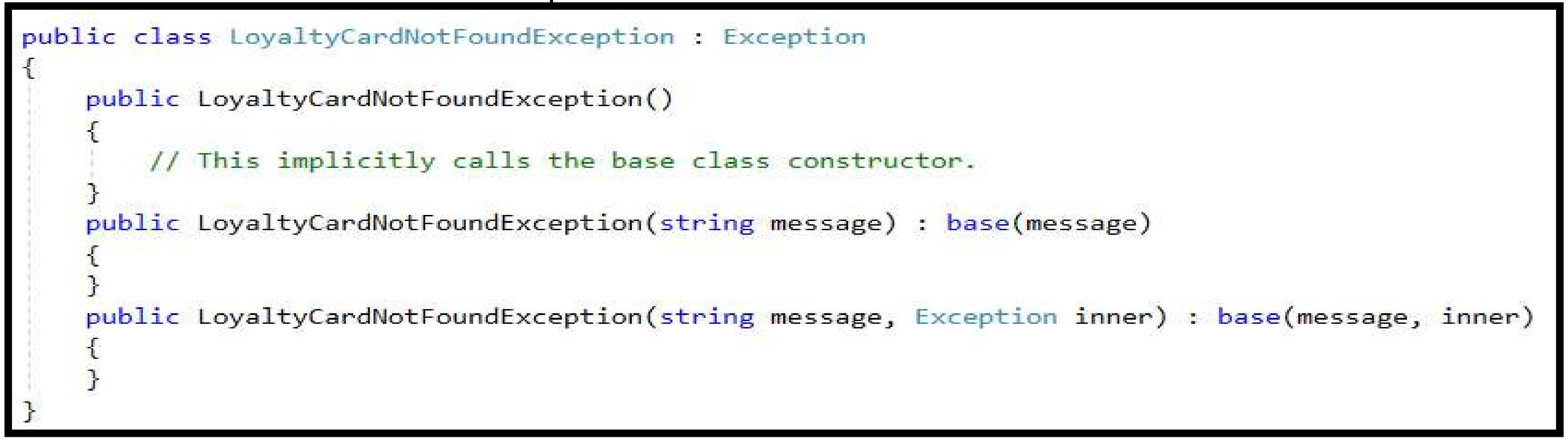
Creating Custom Exceptions

Make use of the **System.Exception** properties:

- The **Message** property enables you to provide more information about what happened as a text string.
- The **InnerException** property enables you to identify another Exception instance that caused the current instance.
- The **Source** property enables you to specify the item or application that caused the error condition.
- The **Data** property enables you to provide more information about the error condition as key-value pairs

To create a custom exception type:

1. Inherit from the **System.Exception** class
2. Implement three standard constructors:
   - base()
   - base(string message)
   - base(string message, Exception inner)
3. Add additional members if required



```
public class LoyaltyCardNotFoundException : Exception
{
    public LoyaltyCardNotFoundException()
    {
        // This implicitly calls the base class constructor.
    }
    public LoyaltyCardNotFoundException(string message) : base(message)
    {
    }
    public LoyaltyCardNotFoundException(string message, Exception inner) : base(message, inner)
    {
    }
}
```

Throwing and Catching Custom Exceptions

- Use the **throw** keyword to throw a custom exception

```
throw new LoyaltyCardNotFoundException();
```

- Use a try/catch block to catch the exception

```
try
{
   // Perform the operation that could cause the exception.
}
catch(LoyaltyCardNotFoundException ex)
{
   // Use the exception variable, ex, to get more information.
}
```

- you should always catch most specific exceptions first and the most general exception (typically System.Exception) last.

Inheriting from Generic Types

- When you inherit from a generic class, you must decide how you want to manage the type parameters of the base class.

- You can handle type parameters in two ways:
  - Leave the type parameter of the base type unspecified.

```
public class CustomList<T> : List<T>
```

  - Specify a type argument for the base type.

```
public class CustomList : List<int>
```

- Regardless of how many—if any—type parameters the base type includes, you can add additional type parameters to your derived class declarations.

```
// Pass the base type parameter on to the derived class, and add an additional type
parameter.
public class CustomCollection1<T, U> : List <T>
// Provide an argument for the base type parameter, but add a new type parameter.
public class CustomCollection2<T> : List<int>
//Inherit from a non-generic class, but add a type parameter.
public class CustomCollection3<T> : CustomBaseClass
```

Creating Extension Methods

- In most cases, if you want to extend the functionality of a class, you use inheritance to create a derived class.

- However, this is not always possible. Many built-in types are sealed to prevent inheritance. For example, you cannot create a class that extends the **System.String** type.
  - As an alternative to using inheritance to extend a type, you can **create extension methods**.
  - When you create extension methods, you are creating methods that you can call on a particular type without actually modifying the underlying type.
  - An extension method is a type of static method.
  - To create an extension method, you create a static method within a static class.
    - The first parameter of the method specifies the type you want to extend.
    - By preceding the parameter with the **this** keyword, you indicate to the compiler that your method is an extension method to that type.

```
public static bool ContainsNumbers(this string s) {...}
```

- Call the method like a regular instance method
  - To use an extension method, you must explicitly import the namespace that contains your extension method by using a using directive

```
string text = "Text with numb3r5 ";
if(text.ContainsNumbers)
{
   // Do something.
}
```

Module Review and Takeaways

- Review Question(s)

- **Question**: Which of the following types of method must you implement in derived classes?
  - (   )Option 1: Abstract methods.
  - (   )Option 2: Protected methods.
  - (   )Option 3: Public methods.
  - (   )Option 4: Static methods.
  - (   )Option 5: Virtual methods.

- **Question:** You want to create an extension method for the **String** class. You create a static method within a static class. How do you indicate that your method extends the **String** type?
  - (   )Option 1: The return type of the method must be a String.
  - (   )Option 2: The first parameter of the method must be a String.
  - (   )Option 3: The class must inherit from the String class.
  - (   )Option 4: The method declaration must include String as a type argument.
  - (   )Option 5: The method declaration must be preceded by String.