Module Overview

Module Overview

- Creating Class Hierarchies
- Extending .NET Framework Classes

Overview

The concept of inheritance is central to object-oriented programming in any language. It is also one of the most powerful tools in your developer toolbox. Essentially, inheritance enables you to create new classes by inheriting characteristics and behaviors from existing classes. When you inherit from an existing class and add some functionality of your own, your class becomes a more specialized instance of the existing class. 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.

In this module, you will learn how to use inheritance to create class hierarchies and to extend .NET Framework types.

Objectives

After completing this module, you will be able to:

- Create base classes and derived classes by using inheritance.
- Create classes that inherit from .NET Framework classes.

Lesson 1: Creating Class Hierarchies

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

Lesson Overview

Rather than creating new classes from nothing, in many cases you can use an existing class as a base for your new class. This is known as inheritance. Your class inherits all the members from the base class, and you simply include

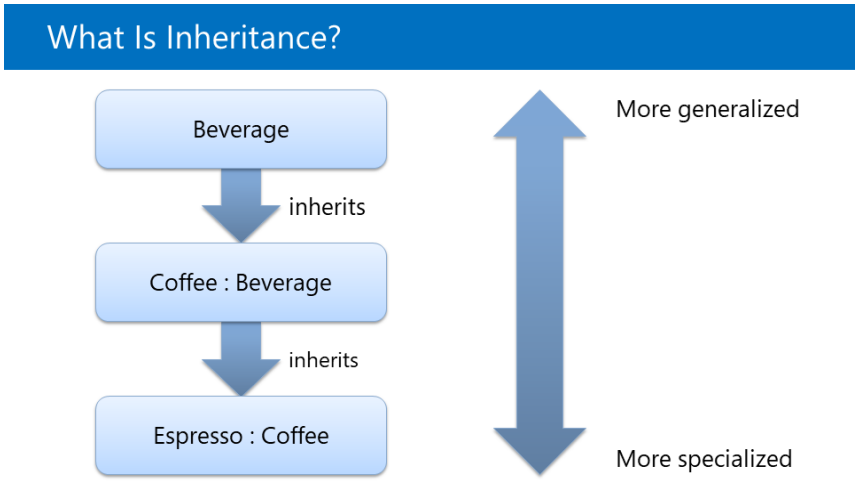
the functionality that you want to add to the base class's capabilities. This way, your class becomes a more specialized version of the base class. This concept of inheritance is one of the main pillars of object-oriented programming.
In this lesson, you will learn how to use inheritance to create class hierarchies.

OBJECTIVES

After completing this lesson, you will be able to:

- Describe inheritance.
- Create base classes.
- Create base class members.
- Create classes that inherit from base classes.
- Call base class methods and constructors from within a derived class.

What Is Inheritance?



The diagram shows a class hierarchy where a class named **Espresso** inherits from a class named **Coffee**, which in turn inherits from a class named **Beverage**. The inherited classes are increasingly specialized instances of the base class.

In Visual C#, a class can *inherit* from another class. When you create a class that inherits from another class, your class is known as the *derived class* and the class that you inherit from is known as the *base class*. The derived class inherits all the members of the base class, including constructors, methods, properties, fields, and events.
Inheritance enables you to build hierarchies of progressively more specialized classes. Rather than creating a class from nothing, you can inherit from a more general base class to provide a starting point for your functionality. Inheritance can help to simplify maintenance of your code.
For example, you define a class named **Beverage**, as shown below:

The Beverage Class

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

Now you want to create a class to represent coffees. Coffee is a type of beverage. It shares all the characteristics and behaviors of a beverage. Rather than creating a class from scratch to represent coffees, you can create a class that inherits from the **Beverage** class. The derived class inherits all the members of the Beverage class, such as the **servingTemperature** field, the **Name** property, the **IsFairTrade** property, and the **GetServingTemperature** method. Within the derived class, you just need to add members that are specific to coffees.
The following example shows how to create a class for coffees that inherits from the **Beverage** class:

The Coffee Class

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

Note: In object-oriented programming, the terms derives and inherits are used interchangeably. Saying that the **Coffee** class derives from the **Beverage** class means the same as saying the **Coffee** class inherits from the **Beverage** class.

As you can see in the previous examples, the syntax for inheriting from a class is similar to the syntax for implementing an interface. This is because inheriting from a class and implementing an interface are both examples of inheritance. However, you do not need to duplicate the base class members in the derived class. By adding the base class to your class declaration, you make all the members of the base class available to consumers of your derived class.

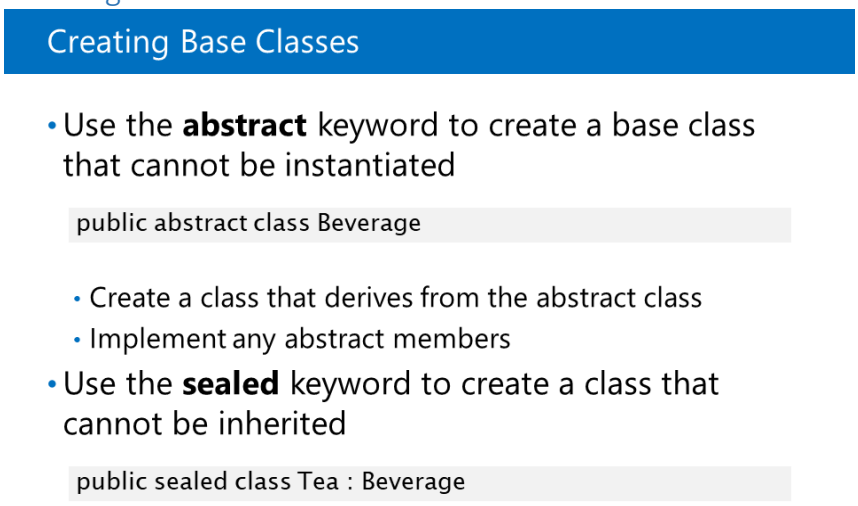
The following example shows how to use base class members in a derived class:

Calling Base Class Members

```
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";
```

As shown in the above example, you can call members of a base class in the same way that you call members of the class itself.

Creating Base Classes



When you create a class, you should consider whether you or other developers will need to use it as a base for derived classes. You have full control over whether, and how, your class can be inherited.

Abstract Classes and Members

As part of an object-oriented design, you may want to create classes that serve solely as base classes for other types. The base class may contain missing or incomplete functionality, and you may not want to enable code to instantiate your class directly. In these scenarios, you can add the **abstract** keyword to your class declaration.

The following example shows how to declare an abstract class:

Declaring an Abstract Class

```
abstract class Beverage
{
}
```

The **abstract** keyword indicates that the class cannot be instantiated directly; it exists solely to define or implement members for derived classes. If you attempt to instantiate an abstract class you will get an error when you build your code.

An abstract class can contain both abstract and non-abstract members. Abstract members are also declared with the **abstract** keyword and are conceptually similar to interface members, in that abstract members define the signature of the member but do not provide any implementation details. 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 following example illustrates the difference between abstract and non-abstract members:

Abstract and Non-Abstract Members

```
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();
}
```

Note: You can only include abstract members within abstract classes. A non-abstract class cannot include abstract members.

Sealed Classes

You may want to create classes that cannot be inherited from. You can prevent developers from inheriting from your classes by marking the class with the **sealed** keyword.

The following example shows how to use the sealed modifier:

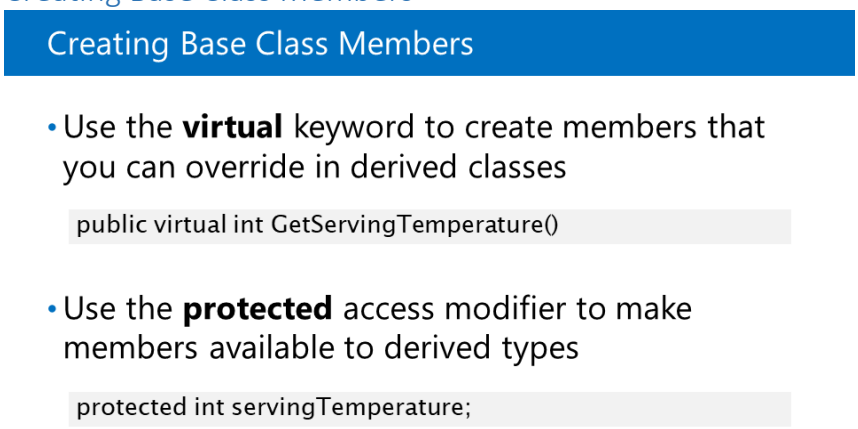
Creating a Sealed Class

```
public sealed class Tea : Beverage
{
// …
}
```

You can apply the **sealed** modifier to classes that inherit from other classes and classes that implement interfaces. You cannot use the **sealed** modifier and the **abstract** modifier on the same class, as a sealed class is the opposite of an abstract class—a sealed class cannot be inherited, whereas an abstract class must be inherited.

Any static class is also a sealed class. You can never inherit from a static class. Similarly, any static members within non-static classes are not inherited by derived classes.

Creating Base Class Members



You may want to implement a method in your base class, but allow derived classes to replace your method implementation with more specific functionality. To create a member that developers can override in a derived class, you use the **virtual** keyword.

The following example shows how to create a virtual method in a class:

Adding Virtual Members

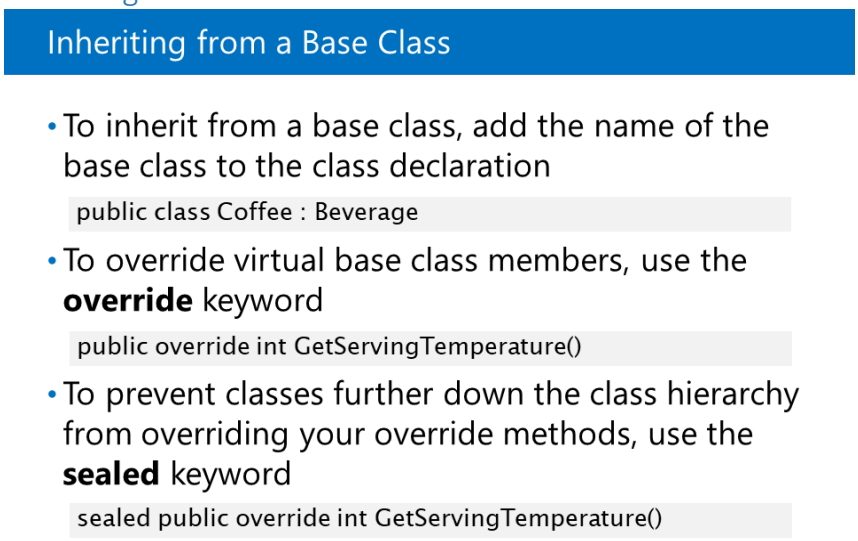
```
public class Beverage
{
protected int servingTemperature;
public string Name { get; set; }
public bool IsFairTrade { get; set; }
public virtual int GetServingTemperature()
{
// This is the default implementation of the GetServingTemperature method.
// Because the method is declared virtual, you can override the implementation in derived classes.
return servingTemperature;
}
}
```

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

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.
private protected	The member is available to types derived from the containing class, but only within its containing assembly. (Only available since Visual C# 7.2)

Members of a class are private by default. Private members are not inherited by derived classes. If you want members that would otherwise be private to be accessible to derived classes, you must prefix the member with the **protected** keyword.

Inheriting from a Base Class



To inherit from another class, you add a colon followed by the name of the base class to your class declaration. The following example shows how to inherit from a base class:

Declaring a Class that Inherits From a Base Class

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

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. However, your class can implement one or more interfaces in addition to deriving from a base type.

Overriding Base Class Members

In some cases you may want to change the way a base class member works in your derived class.

For example, the **Beverage** base class includes a method named **GetServingTemperature**:

Adding Virtual Members

```
public class Beverage
{
protected int servingTemperature;
public virtual int GetServingTemperature()
{
return servingTemperature;
```

```
}
// Other class members not shown.
}
```

Because **GetServingTemperature** is a virtual method, you can override it in a derived class. To override a virtual method in a derived class, you create a method with the same signature and prefix it with the override keyword.
The following example shows how to override a virtual method in a derived class:

Overriding a Virtual Method by Using the override Keyword

```
public class Coffee : Beverage
{
protected bool includesMilk;
private int servingTempWithMilk;
private int servingTempWithoutMilk;
public override int GetServingTemperature()
{
if(includesMilk) return servingTempWithMilk
else return servingTempWithoutMilk;
}
}
```

You can use the same approach to override properties, indexers, and events. In each case, you can only override a base class member if the member is marked as virtual in the base class. You cannot override constructors.
You can also override a base class member by using the **new** keyword:

Overriding a Virtual Method by Using the new Keyword

```
public class Coffee : Beverage
{
public new int GetServingTemperature()
{
// …
}
}
```

When you use the **override** keyword, your method *extends* the base class method. By contrast, when you use the **new** keyword, your method *hides* the base class method. This causes subtle but important differences in how the compiler treats your base class and derived class types.
Note: Override can be used for both **virtual** and **abstract** members. The difference is that overriding virtual members is optional, while overriding abstract members is mandatory. Only an **abstract** class must implement every **abstract** member defined by its ancestors.
Reference Links: For a detailed explanation of the differences in behavior between the override keyword and the new keyword, see Knowing When to Use Override and New Keywords (C# Programming Guide) at https://aka.ms/moc-20483c-m5-pg1.

Sealing Overridden Members

When you override a base class member, you can prevent classes that derive from your class from overriding your implementation of the base class member by using the **sealed** keyword.
The following example shows how to seal an overridden base class member:

Sealing an Overridden Member

```
public class Coffee : Beverage
{
sealed public override int GetServingTemperature()
{
// Derived classes cannot override this method.
}
}
```

By sealing an overridden member, you force any classes that derive from your class to use your implementation of the base class member, rather than creating their own. This can be useful when you need to control the behavior of your classes and ensure that derived classes do not attempt to modify how specific members work.
You can only apply the **sealed** modifier to a member if the member is an **override** member. Remember that members are inherently sealed unless they are marked as **virtual**. In this case, because 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.

Calling Base Class Constructors and Members

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

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

You can use the **base** keyword to access base class methods and constructors from within a derived class. This is useful in the following scenarios:

- You override a base class method, but you still want to execute the functionality in the base class method in addition to your own additional functionality.
- You create a new method, but you want to call a base class method as part of your method logic.
- You create a constructor and you want to call a base class constructor as part of your initialization logic.
- You want to call a base class method from a property accessor.

Calling Base Class Constructors from a Derived Class

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. However, in some circumstances, you might want your constructors to call an alternative base class constructor. In these cases, you can use the **base** keyword in your constructor declarations to specify which base class constructor you want to call.

The following example shows how to call base class constructors:

Calling Base Class Constructors

```
public class Beverage
{
public Beverage()
{
// Implementation details not shown.
}
public Beverage(string name, bool isFairTrade, int servingTemp)
{
// Implementation details are not shown.
}
// Other class members are not shown.
}
public class Coffee : Beverage
{
public Coffee()
{
// This constructor implicitly calls the default Beverage constructor.
// The declaration is implicitly equivalent to public Coffee() : base()
// You can include additional code here.
}
public Coffee(string name, bool isFairTrade, int servingTemp, string bean, string roast)
: base(name, isFairTrade, servingTemp)
{
// This calls the Beverage(string, bool, int) constructor.
// You can include additional code here:
Bean = bean;
Roast = roast;
}
public string Bean { get; set; }
public string Roast { get; set; }
public string CountryOfOrigin { get; set; }
```

```
}
```

As the previous example shows, you must call the base class constructor from your constructor declaration. You cannot call the base class constructor from within your constructor method body. You can use the named parameters from your constructor declaration as arguments to the base class constructor.

Calling Base Class Methods from a Derived Class

You can call base class methods from within method bodies or property accessors in a derived class. To do this, you use the **base** keyword as you would use an instance variable.

The following example shows how to call base class methods:

Calling Base Class Methods

```
public class Beverage
{
protected int servingTemperature;
public virtual int GetServingTemperature()
{
return servingTemperature;
}
// Constructors and additional class members are not shown.
}
public class Coffee : Beverage
{
bool iced = false;
protected int servingTempIced = 40;
public override int GetServingTemperature()
{
if(iced)
{
return servingTempIced;
}
else
{
return base.GetServingTemperature();
}
}
}
```

Remember that the rules of inheritance do not apply to static classes and members. As such, you cannot use the **base** keyword within a static method.

Demonstration: Calling Base Class Constructors

Demonstration: Calling Base Class Constructors

In this demonstration, you will learn how to:

- Create a derived class constructor that calls a default base class constructor implicitly
- Create a derived class constructor that calls a specific base class constructor explicitly
- Observe the order of constructor execution as a derived class is instantiated

In this demonstration, you will step through the execution of an application. The solution includes a class named **Coffee** that inherits from a class named **Beverage**. The **Coffee** class includes two constructors—one that implicitly calls the default base class constructor, and one that explicitly calls an alternative base class constructor. The application creates instances of the **Coffee** class by using both of these constructors. In both cases, you can observe how derived class constructors call base class constructors. You will also see how derived class constructors pass argument values to base class constructors.

Demonstration Steps

You will find the steps in the **Demonstration: Calling Base Class Constructors** section on the following page: https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD05_DEMO.md.

Lesson 2: Extending .NET Framework Classes

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

Lesson Overview

The .NET Framework contains several thousand classes that provide a wide range of functionality. When you create your own classes, you should look to build on these classes by inheriting from .NET Framework types wherever possible. Not only does this reduce the amount of code you need to write, it also helps to ensure that your classes work in a standardized way.

The .NET Framework also enables you to create *extension methods* to add functionality to sealed .NET Framework types. This enables you to extend the functionality of built-in types, such as the **String** class, when the inheritance approach is not permitted.

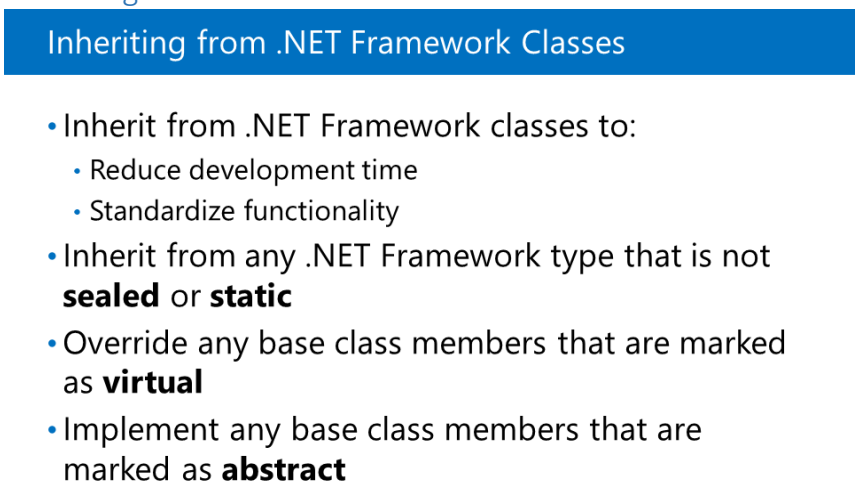
In this lesson, you will learn how to extend .NET Framework types by using inheritance and extension methods.

OBJECTIVES

After completing this lesson, you will be able to:

- Create classes that inherit from .NET Framework types.
- Create custom exception classes.
- Throw and catch custom exceptions.
- Create classes that inherit from generic types.
- Create extension methods for .NET Framework types.

Inheriting from .NET Framework Classes



There are almost 15,000 public types in the .NET Framework. Although not all of these are extendable classes, many of them are. When you want to develop a class, in many cases there is a built-in .NET Framework class that can provide a foundation for your code.

There are two key advantages to creating a class that inherits from a .NET Framework class, rather than developing a class from scratch:

- **Reduced development time**. By inheriting from an existing class, you reduce the amount of logic that you have to create yourself.
- **Standardized functionality**. Just like implementing an interface, inheriting from a standard base class means that your class will work in a standardized way. You can also represent instances of your class as instances of the base class, which makes it easier for developers to use your class alongside other types that derive from the same base class.

The rules of inheritance apply to built-in .NET Framework classes in the same way they apply to custom classes:

- You can create a class that derives from a .NET Framework class, providing that the class is not *sealed* or *static*.
- You can 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.

When you create a class, select a base class that minimizes the amount of coding and customization required. If you find yourself replicating functionality that is available in built-in classes, you should probably choose a more specific base class. On the other hand, if you find that you need to override several members, you should probably choose a more general base class.

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. In addition, you can take advantage of the **Sort** method in the **List<T>** class. If you call the **Sort** method, any duplicate items will be adjacent in the collection, which can make it easier to identify and remove them.

The following example shows how to extend the **List<T>** class:

Extending a .NET Framework Class

```
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);
}
}
}
}
```

When you use this approach, consumers of your class have access to all the functionality provided by the base **List<T>** class. They also have access to the additional **RemoveDuplicates** method that you provided in your derived class.

Creating Custom Exceptions

Creating Custom Exceptions

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

The .NET Framework contains built-in exception classes to represent most common error conditions. 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**.

Note: Most built-in exception classes are defined in the System namespace. For more information about the System namespace, see the System Namespace page at https://aka.ms/moc-20483c-m5-pg2.

When you need to throw exceptions in your code, you should reuse existing .NET Framework exception types wherever possible. However, there may be circumstances when you want to create your own custom exception types.

When Should You Create a Custom Exception Type?

You should consider creating a custom exception type when:

- 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.

Remember that the primary purpose of exception types is to enable you to handle specific error conditions in specific ways by catching a specific exception type. The exception type is not designed to communicate the precise details of the problem. All exception classes include a message property for this purpose. Therefore, you should not create a custom exception class just to communicate the nature of an error condition. Create a custom exception class only if you need to handle that error condition in a distinct way.

Creating Custom Exception Types

All exception classes ultimately derive from the **System.Exception** class. This class provides a range of properties that you can use to provide more detail about the error condition. For example:

- 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.

When you create a custom exception type, you should make use of these existing properties wherever possible, rather than creating your own alternative properties. At a high level, the process for creating a custom exception class is as follows:

1. Create a class that inherits from the **System.Exception** class.
2. Map your class constructors to base class constructors.
3. Add any members if required.

The following example shows how to create a custom exception class:

Creating a Custom Exception Type

```
using System;
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)
{
}
}
```

Note: When you create a custom exception class, it is a best practice to include the word Exception at the end of your class name.

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.
}
```

After you have created your custom exception type, you can throw and catch custom exceptions in the same way that you would throw and catch any other exceptions. To throw a custom exception, you use the **throw** keyword and create a new instance of your exception type.

The following code shows how you can throw a custom exception:

Throwing a Custom Exception

```
public LoyaltyCard
{
public static int GetBalance(string loyaltyCardNumber)
{
var customer = LoyaltyCard.GetCustomer(loyaltyCardNumber);
if(customer == null)
{
throw new LoyaltyCardNotFoundException("The card number provided was not found");
}
else
{
return customer.TotalPoints;
}
}
// Other class members are not shown.
}
```

To catch the exception, you use a try/catch block. Remember that you should always attempt to catch the most specific exceptions first, and the most general exception (typically **System.Exception**) last.

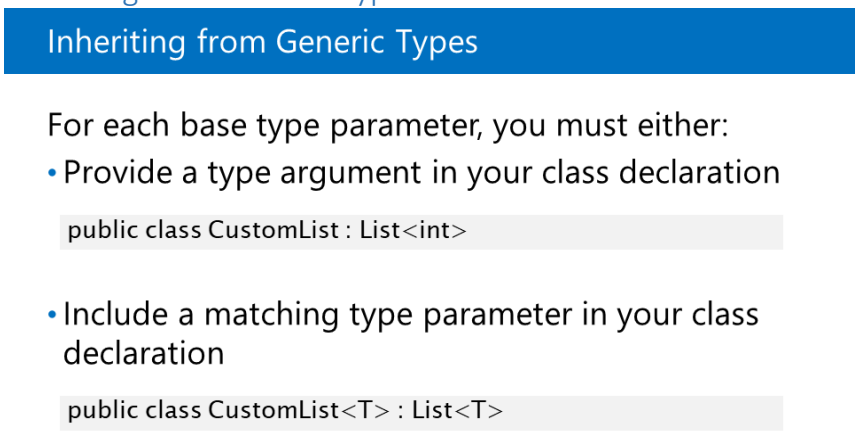
The following example shows how you can catch a custom exception:

Catching a Custom Exception

```
public bool PayWithPoints(int costInPoints, string cardNumber)
try
{
int totalPoints = LoyaltyCard.GetBalance(cardNumber);
// Throws a LoyaltyCardNotFoundException if the card number is invalid.
if(totalPoints >= costInPoints)
{
LoyaltyCard.DeductPoints(costInPoints);
```

```
return true;
}
else return false;
}
catch(LoyaltyCardNotFoundException)
{
// Take appropriate action to remedy the invalid card number.
return false;
}
catch(Exception)
{
// Catches other unanticipated exceptions.
return false;
}
```

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.
- Specify a type argument for the base type.

Consider an example where you want to create a custom list class that inherits from List<T>. If you leave the type parameter of the base type unspecified, you must include the same type parameter in your class declaration.
The following example shows how to inherit from a generic base type without specifying a type argument:

Inheriting from a Generic Base Type Without Specifying a Type Argument

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

In the above example, when you instantiate the **CustomList** class and provide a type argument for **T**, the same type argument is applied to the base class.
Alternatively, you can specify a type argument for the base type in your class declaration. When you use this approach, any references to the type parameter in the base type are replaced with the type you specify in your class declaration.
The following example shows how to specify a type argument for a base type:

Inheriting from a Generic Base Type by Specifying a Type Argument

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

In the above example, when you instantiate the **CustomList** class, you do not need to specify a type parameter. Any base class methods or properties that referenced the type parameter are now strongly typed to the **Int32** type. For example, the **List.Add** method will only accept arguments of type **Int32**.
If the base class that you are inheriting from contains multiple type parameters, you can specify type arguments for any number of them. The important thing to remember is that you must *either* provide a type argument *or* add a matching type parameter to your class declaration for each type parameter in the base type.
The following example shows the different ways in which you can inherit from a base type with multiple type parameters:

Inheriting from a Base Type with Multiple Type Parameters

```
// Pass all the base type parameters on to the derived class.
```

```
public class CustomDictionary1<TKey, TValue> : Dictionary<TKey, TValue> { }
// Provide an argument for one of the base type parameters and pass the other one to the derived class.
public class CustomDictionary2<TValue> : Dictionary<int, TValue> { }
// Provide arguments for both of the base type parameters.
public class CustomDictionary3 : Dictionary <int, string> { }
```

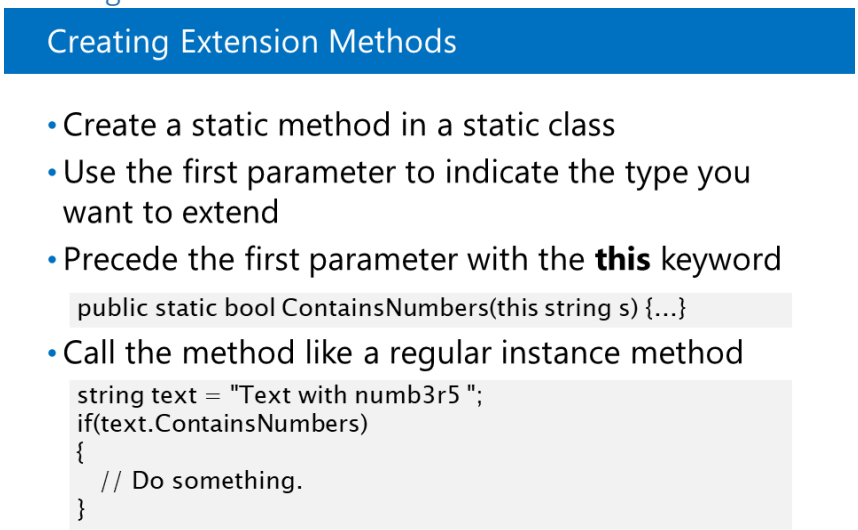
Regardless of how many—if any—type parameters the base type includes, you can add additional type parameters to your derived class declarations.

The following example shows how to add additional type parameters to derived class declarations:

Adding Type Parameters to 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.

The following example shows how to create an extension method for the **System.String** type:

Creating an Extension Method

```
namespace FourthExtensionMethods;
{
public static class FourthCoffeeExtensions
{
public static bool ContainsNumbers(this String s)
{
// Use regular expressions to determine whether the string contains any numerical digits.
return Regex.IsMatch(s, @"\d");
}
}
}
```

To use an extension method, you must explicitly import the namespace that contains your extension method by using a **using** directive:

Bringing an Extension Method Into Scope

```
using FourthExtensionMethods;
```

You can then call the extension method as if it was an instance method on the type that it extends:

Calling an Extension Method

```
Console.WriteLine("Please type some text that contains numbers and then press Enter");
string text = Console.ReadLine();
if(text.ContainsNumbers)
{
Console.WriteLine("Your text contains numbers. Well done!");
}
else
{
Console.WriteLine("Your text does not contain numbers. Please try again.");
}
```

Demonstration: Refactoring Common Functionality into the User Class Lab

Demonstration: Refactoring Common Functionality into the User Class Lab

In this demonstration, you will learn about the tasks that you will perform in the lab for this module

In this demonstration, you will learn about the tasks that you will perform in the lab for this module.

Demonstration Steps

You will find the steps in the **Demonstration: Refactoring Common Functionality into the User Class Lab** section on the following page: https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD05_DEMO.md.

Lab Scenario

Lab Scenario

You have noticed that the Student and Teacher classes in the Grades application contain some duplicated functionality. To make the application more maintainable, you decide to refactor this common functionality to remove the duplication.

You are also concerned about security. Teachers and students all require a password, but it is important to maintain confidentiality and at the same time ensure that students (who are children) do not have to remember long and complex passwords. You decide to implement different password policies for teachers and students; teachers' passwords must be stronger and more difficult to guess than student passwords.

Also, you have been asked to update the application to limit the number of students that can be added to a class. You decide to add code that throws a custom exception if a user tries to enroll a student in a class that is already at capacity.

Lab: Developing the Class Enrollment Application

Lab: Refactoring Common Functionality into the User Class

- Exercise 1: Creating and Inheriting from the User Base Class
- Exercise 2: Implementing Password Complexity by Using an Abstract Method
- Exercise 3: Creating the ClassFullException Custom Exception

Estimated Time: 60 minutes

Scenario

You have noticed that the Student and Teacher classes in the Grades application contain some duplicated functionality. To make the application more maintainable, you decide to refactor this common functionality to remove the duplication.

You are also concerned about security. Teachers and students all require a password, but it is important to maintain confidentiality and at the same time ensure that students (who are children) do not have to remember long and complex passwords. You decide to implement different password policies for teachers and students; teachers' passwords must be stronger and more difficult to guess than student passwords.

Also, you have been asked to update the application to limit the number of students that can be added to a class. You decide to add code that throws a custom exception if a user tries to enroll a student in a class that is already at capacity.

Objectives

After completing this lab, you will be able to:

- Use inheritance to factor common functionality into a base class.
- Implement polymorphism by using an abstract method.
- Create a custom exception class.

Lab Setup

You will find the high-level steps on the following page: https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD05_LAB_MANUAL.md.

You will find the detailed steps on the following page: https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD05_LAK.md.

Module Review and Takeaways

Module Review and Takeaways

- Review Questions

In this module, you have learned how to use inheritance and extension methods to extend the functionality of existing types.

Review Questions

1. Which of the following types of method *must* you implement in derived classes?

( )Abstract methods.
( )Protected methods.
( )Public methods.
( )Static methods.
( )Virtual methods.

2. 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?

( )The return type of the method must be a String.
( )The first parameter of the method must be a String.
( )The class must inherit from the String class.
( )The method declaration must include String as a type argument.
( )The method declaration must be preceded by String.