Module 06(More OOP)

Introducing Inheritance

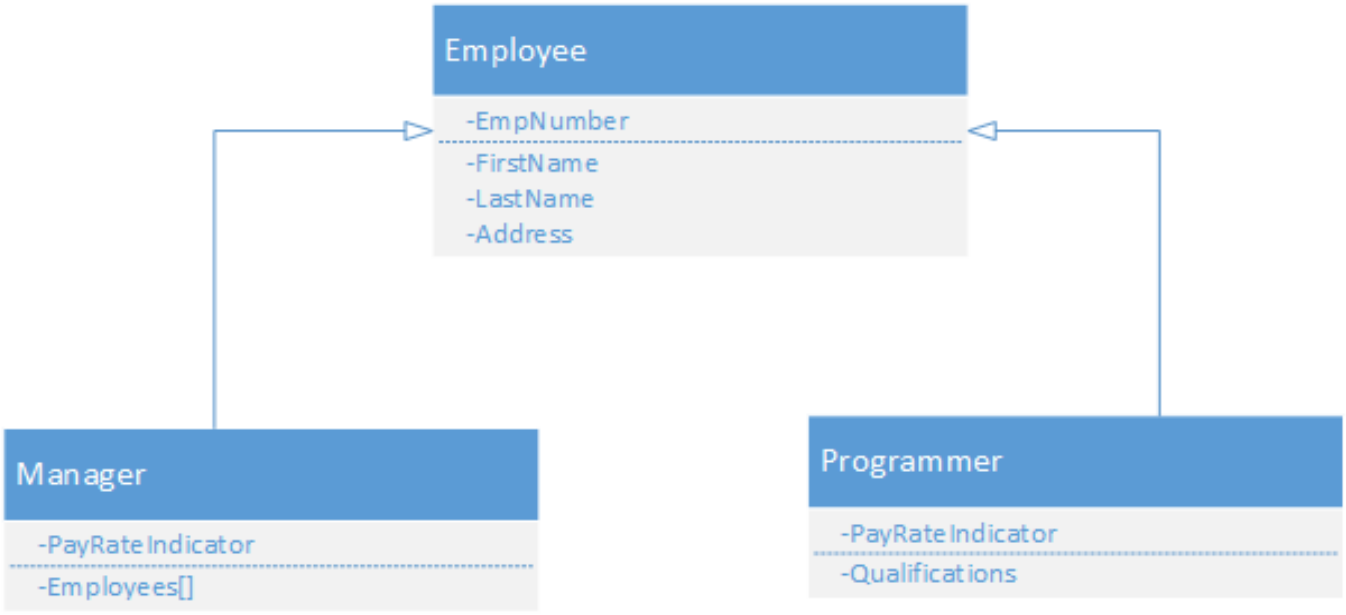
Inheritance is yet another pillar in the world of object-oriented programming. You can use inheritance as an aspect of code reuse by defining different classes that will contain common features and have a relationship to one another. An example could be employees as a general classification and which could contain managers, non-management workers, and any other employee classification.

Consider creating an application to simulate an office workspace that includes all the employees. Then consider the common features that all employee classifications have followed by a list of attributes that are different for each employee type. For example, they all might have an employee number, first and last names, addresses, etc., but managers have different responsibilities than other employee classifications.

Inheritance allows you to create a base class containing the core, shared attributes, and then each different class of employee would inherit these attributes whole extending them for their own special needs. The class that inherits from the base class is referred to as the derived class but also commonly referred to as a subclass. When using the term subclass, some also refer to the base class as a super class. In programming languages such as Objective-C, this is reinforced by the use of statements such like this example where the keyword super is used to initialize a nib file in a super class;

```
self = [super initWithNibName:nibNameOrNil bundle:nibBundleOrNil];
```

Consider the following simplified UML Class diagram as an example.



Applying Inheritance

The C# programming language does not support multiple inheritance directly. Multiple inheritance is a concept whereby multiple base classes can be inherited by a single subclass. In C#, a derived class can only have one base class.

To inherit from base class in C#, you append your derived class name with a colon and the name of the base class. The following example demonstrates the Manager class inheriting the Employee base class from the previous topic's UML diagram.

```
class Manager : Employee
{
   private char payRateIndicator;
   private Employee[] emps;
}
```

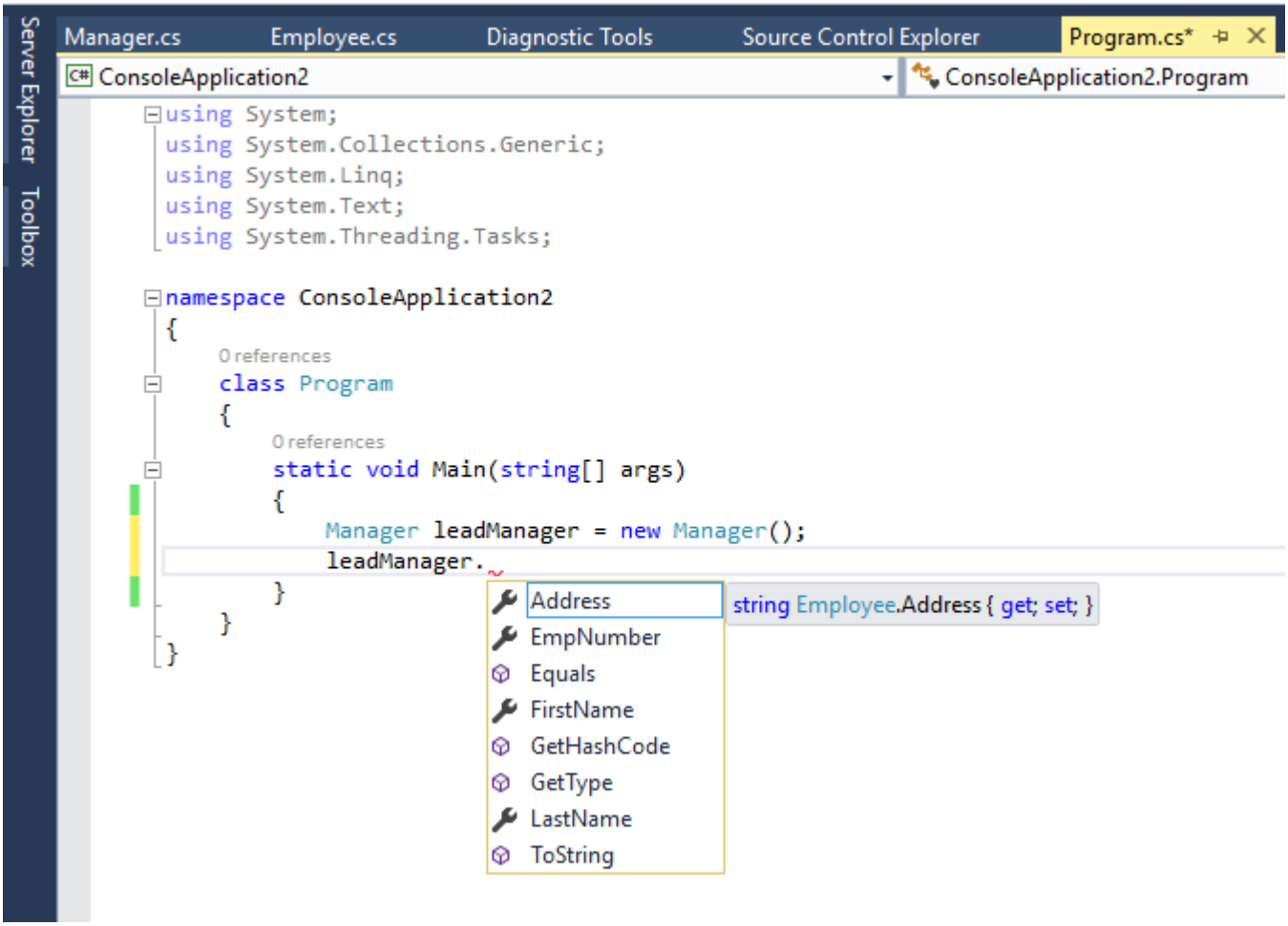
This simple class definition in C# lists the keyword classfollowed by the class name Manager, a colon and then the name of the base class Employee. Looking at this snippet we can't tell what the Manager class has inherited from Employeeso we would need to look at that class as well to understand all the properties available for us. The Employee class is shown here:

```
class Employee
{
   private string empNumber;
   private string firstName;
   private string lastName;
```

```
private string address;

public string EmpNumber
{
   get
   {
      return empNumber;
   }

   set
   {
      empNumber = value;
   }
}

public string FirstName
{
   get
   {
      return firstName;
   }

   set
   {
      firstName = value;
   }
}

public string LastName
{
   get
   {
      return lastName;
   }

   set
   {
      lastName = value;
   }
}

public string Address
{
   get
```

```
        {
            return address;
        }

        set
        {
            address = value;
        }
    }
}
```

When working in Visual Studio, the Intellisense feature will provide you with a visual representation of the inherited members. As an example, if we were to instantiate an object of type Manager in our code and then use the dot notation to bring up the list of properties for the Manager class, we would also see the properties from the base class Employee in that list as well. This is shown in the following image:



Abstract Classes

Looking back at our inheritance topic, we note that theEmployee class is being used as the base class for Managerand Programmer. We can continue to extend the Employeeclass by creating as many sub classes as required for different employees in our application. However, when looking at our class hierarchy, does it make sense to be able to create an object of type Employee, directly? Certainly the base class contains common properties but realistically we would prefer to only create objects of specific types of employees.

In order to enforce this behavior in our code, we should consider making the Employee class an abstract class. Abstract classes are closely related to interfaces, which will be covered in the next topic. Abstract classes cannot be instantiated, which means we would not be able to create a new Employee object in code with this statement:

Employee newEmployee = new Employee();

When you create an abstract class you may partially implement some of the behavior in the class, or not implement the behavior at all. An abstract class requires the subclass to implement some, or all, of the functionality. If we extend our previous example of the Employee and Manager classes, using abstract classes, we can demonstrate this concept better. Note that the employee class now includes some methods to implement behaviors.

```
abstract class Employee
  {
     private string empNumber;
     private string firstName;
     private string lastName;
     private string address;

     .....

     public virtual void Login()
     {
     }

     public virtual void LogOff()
     {
     }

     public abstract void EatLunch();

  }
```

In this code sample, the ..... is used to denote that some code was snipped from this sample to shorten it for display. Consider the properties still present in the code.

Also notice that we have now prepended the keyword abstract to our class: abstract class Employee. Doing so converts our class to an abstract class and sets up some requirements. Once you create an abstract class, you decide which methods "must" be implemented in the sub classes and which methods "can" be implemented, or overridden, in the sub class. There is a clear difference.

Any method you declare in the abstract class that will contain some implementation in the abstract class, but can be overridden in the sub class, you decorate with the virtual keyword. Note in the previous code sample, Login() andLogOff() are both decorated with the virtual keyword. This means that you can write implementation code in the abstract class and sub classes are free to override the implementation , or accept the implementation that is inherited.

The EatLunch() method is decorated with the abstract keyword, like the class. There are specific constraints around an abstract method:

- An abstract method cannot exist in non-abstract class
- An abstract method is not permitted to have any implementation, including curly braces
- An abstract method signature must end in a semi-colon
- An abstract method **MUST** be implemented in any sub class. Failure to do so will generate a compiler warning in C#.

Sealed Classes

We have covered inheritance for class heirarchies and have shown how base classes can be inherited by sub classes and we have dicussed abstract classes. Both concepts focus on the ability of a class to be inherited and provide attributes and behaviors to other classes, for the purpose of code reuse. But what happens if you decide that you don't want your class to be inherited? How do you prevent that from happening? Quite simply, you can create a sealed class. You can use the sealed keyword on your class restrict the inheritance feature of object oriented programming. If a class is derived from a sealed class then the compiler throws an error.

Although we didn't cover this in the topic on structs, it is important to note that while structs are like classes in some aspects, structs are sealed. Therefore you cannot derive a class from a struct.

Introducing Interfaces

An interface is a little bit like a class without an implementation. It specifies a set of characteristics and behaviors by defining signatures for methods, properties, events, and indexers, without specifying how any of these members are implemented. When a class implements an interface, the class provides an implementation for each member of the interface. By implementing the interface, the class is thereby guaranteeing that it will provide the functionality specified by the interface.

Note the important distinction when using an Interface. A class "implements" and interface as opposed to "inheriting" a base class.

Creating Interfaces

You can think of an interface as a contract. By implementing a particular interface, a class guarantees to consumers that it will provide specific functionality through specific members, even though the actual implementation is not part of the contract.

The syntax for defining an interface is similar to the syntax for defining a class. You use the interface keyword to declare an interface, as shown by the following example:

```
// Declaring an Interface
public interface IBeverage
{
  // Methods, properties, events, and indexers go here.
}
```

Note: Programming convention dictates that all interface names should begin with an "I".

Similar to a class declaration, an interface declaration can include an access modifier. You can use the following access modifiers in your interface declarations:

Access modifier	Description
public	The interface is available to code running in any assembly.
internal	The interface is available to any code within the same assembly, but not available to code in another assembly. This is the default value if you do not specify an access modifier.

Adding Interface Members

An interface defines the signature of members but does not include any implementation details. Interfaces can include methods, properties, events, and indexers:

- To define a method, you specify the name of the method, the return type, and any parameters:
  int GetServingTemperature(bool includesMilk);
- To define a property, you specify the name of the property, the type of the property, and the property accessors:
  bool IsFairTrade { get; set; }
- To define an event, you use the event keyword, followed by the event handler delegate, followed by the name of the event:
  event EventHandler OnSoldOut;
- To define an indexer, you specify the return type and the accessors:
  string this[int index] { get; set; }

Interface members do not include access modifiers. The purpose of the interface is to define the members that an implementing class should expose to consumers, so that all interface

members are public. Interfaces cannot include members that relate to the internal functionality of a class, such as fields, constants, operators, and constructors.

Let's see a concrete example. Suppose that you want to develop a loyalty card scheme for an application related to a coffee company. You might start by creating an interface named ILoyaltyCardHolder that defines:

- A read-only integer property named TotalPoints.
- A method named AddPoints that accepts a decimal argument.
- A method named ResetPoints.

The following example shows an interface that defines one read-only property and two methods:

```
// Defining an Interface
public interface ILoyaltyCardHolder
{
  int TotalPoints { get; }
  int AddPoints(decimal transactionValue);
  void ResetPoints();
}
```

Notice that the methods in the interface do not include method bodies. Similarly, the properties in the interface indicate which accessors to include but do not provide any implementation details. The interface simply states that any implementing class must include and provide an implementation for the three members. The creator of the implementing class can choose how the methods are implemented. For example, any implementation of the AddPoints method will accept a decimal argument (the cash value of the customer transaction) and return an integer (the number of points added). The class developer could implement this method in a variety of ways. For example, an implementation of the AddPoints method could:

- Calculate the number of points to add by multiplying the transaction value by a fixed amount.
- Get the number of points to add by calling a service.
- Calculate the number of points to add by using additional factors, such as the location of the loyalty cardholder.

The following example shows a class that implements the ILoyaltyCardHolder interface:

```
// Implementing an Interface
public class Customer : ILoyaltyCardHolder
{
  private int totalPoints;
  public int TotalPoints
  {
    get { return totalPoints; }
  }
  public int AddPoints(decimal transactionValue)
  {
    int points = Decimal.ToInt32(transactionValue);
    totalPoints += points;
  }
  public void ResetPoints()
  {
    totalPoints = 0;
  }
  // Other members of the Customer class.
}
```

The details of the implementation do not matter to calling classes. By implementing the ILoyaltyCardHolder interface, the implementing class is indicating to consumers that it will take care of the AddPoints operation. One of the key advantages of interfaces is that they enable you to modularize your code. You can change the way in which your class implements the interface at any point, without having to update any consumer classes that rely on an interface implementation.

Implicit and Explicit Implementation

When you create a class that implements an interface, you can choose whether to implement the interface implicitly or explicitly. To implement an interface implicitly, you implement each interface member with a signature that matches the member definition in the interface. To implement an interface explicitly, you fully qualify each member name so that it is clear that the member belongs to a particular interface.

The following example shows an explicit implementation of the IBeverage interface:

```
// Implementing an Interface Explicitly
public class Coffee : IBeverage
{
  private int servingTempWithoutMilk { get; set; }
  private int servingTempWithMilk { get; set; }
  public int IBeverage.GetServingTemperature(bool includesMilk)
```

```
  {
    if(includesMilk)
    {
       return servingTempWithMilk;
    }
    else
    {
      return servingTempWithoutMilk;
    }
  }
  public bool IBeverage.IsFairTrade { get; set; }
  // Other non-interface members.
}
```

In most cases, whether you implement an interface implicitly or explicitly is an aesthetic choice. It does not make a difference in how your class compiles. Some developers prefer explicit interface implementation because doing so can make the code easier to understand. The only scenario in which you must use explicit interface implementation is if you are implementing two interfaces that share a member name. For example, if you implement interfaces named IBeverage andIInventoryItem, and both interfaces declare a Boolean property named IsAvailable, you would need to implement at least one of the IsAvailable members explicitly. In this scenario, the compiler would be unable to resolve theIsAvailable reference without an explicit implementation

Interface Polymorphism

As it relates to interfaces, polymorphism states that you can represent an instance of a class as an instance of any interface that the class implements. Interface polymorphism can help to increase the flexibility and modularity of your code. Suppose you have several classes that implement an IBeverage interface, such as Coffee, Tea, Juice, and so on. You can write code that works with any of these classes as instances of IBeverage, without knowing any details of the implementing class. For example, you can build a collection of IBeverage instances without needing to know the details of every class that implements IBeverage.

For example, if the Coffee class implements the IBeverage interface, you can represent a new Coffee object as an instance of Coffee or an instance of IBeverage:

```
// Representing an Object as an Interface Type
Coffee coffee1 = new Coffee();
IBeverage coffee2 = new Coffee();
```

You can use an implicit cast to convert to an interface type, because you know that the class must include all the interface members.

```
// Casting to an Interface Type
IBeverage beverage = coffee1;
```

You must use an explicit cast to convert from an interface type to a derived class type, as the class may include members that are not defined in the interface.

```
// Casting an Interface Type to a Derived Class Type
Coffee coffee3 = beverage as Coffee;
// OR
Coffee coffee4 = (Coffee)beverage;
```

Implementing Multiple Interfaces

In many cases, you will want to create classes that implement more than one interface. For example, you might want to:

- Implement the IDisposable interface to enable the .NET runtime to dispose of your class correctly.
- Implement the IComparable interface to enable collection classes to sort instances of your class.
- Implement your own custom interface to define the functionality of your class.

To implement multiple interfaces, you add a comma-separated list of the interfaces that you want to implement to your class declaration. Your class must implement every member of every interface you add to your class declaration. The following example shows how to create a class that implements multiple interfaces:

```
// Declaring a Class that Implements Multiple Interfaces
public class Coffee: IBeverage, IInventoryItem
{
}
```

Object Lifecycle

The life cycle of an object has several stages, which start at creation of the object and end in its destruction. To create an object in your application, you use the new keyword. When the common language runtime (CLR) executes code to create a new object, it performs the following steps:

1. It allocates a block of memory large enough to hold the object.

2. It initializes the block of memory to the new object.

   The CLR handles the allocation of memory for all managed objects. However, when you use unmanaged objects, you may need to write code to allocate memory for the unmanaged objects that you create. Unmanaged objects are those that are not .NET components such as a Microsoft Word object, a database connection, or a file resource.

   When you have finished with an object, you can dispose of it to release any resources, such as database connections and file handles, that it consumed. When you dispose of an object, the CLR uses a feature called the garbage collector (GC) to perform the following steps:

1. The GC releases resources.
2. The memory that is allocated to the object is reclaimed.

   The GC runs automatically in a separate thread. When the GC runs, other threads in the application are halted, because the GC may move objects in memory and therefore must update the memory pointers.

   Introducing Garbage Collection

   The garbage collector is a separate process that runs in its own thread whenever a managed code application is running. The garbage collection process provides the following benefits:

- Enables you to develop your application without having to worry about freeing memory.
- Allocates objects on the managed heap efficiently.
- Reclaims objects that are no longer being used, clears their memory, and keeps the memory available for future allocations. Managed objects automatically get clean content to start with, so their constructors do not have to initialize every data field.
- Provides memory safety by making sure that an object cannot use the content of another object.

  When a .NET application is executed, the garbage collector is initialized by the CLR. The GC allocates a segment of memory that it will use to store and manage the objects for each .NET application that is running. This memory area is referred to as the managed heap, which differs from a native heap used in the context of the operating system.

  There is a managed heap for each managed process that is running and all threads in the process allocate memory for objects, in that process, on the same heap. This means that each process has its own virtual memory space.

To reserve memory, the garbage collector calls the Win32 VirtualAlloc function, and reserves one segment of memory at a time for managed applications. The garbage collector also reserves segments as needed, and releases segments back to the operating system (after clearing them of any objects) by calling the Win32 VirtualFree function.

Note: The size of segments allocated by the garbage collector is implementation-specific and is subject to change at any time, including in periodic updates. When writing your app, you should never make assumptions about, or depend on a particular segment size that will be used by the GC.

When a garbage collection is triggered, the process will reclaim memory that is occupied by dead objects, objects no longer referenced in the application code. Reclaiming also compacts live objects so that they are moved together, dead space is removed, which reduces the size of the heap.

The GC does exact a performance hit on the applications because garbage collection is the result of the number of allocations and the amount of memory usage and release on the managed heap.

Garbage collection occurs when one of the following conditions is true:

- The system is running low on physical memory.
- The memory that is used by currently allocated objects surpasses an acceptable threshold. This threshold will be continuously adjusted as the process is running.
- The GC.Collect method is called. While you can call this method yourself, typically you do not have to call this method, because the garbage collector runs continuously. Even if you do call this method, there is no guarantee that it will run precisely when you call it.

mplementing the Dispose Pattern

The dispose pattern is a design pattern that frees resources that an object has used. The .NET Framework provides theIDisposable interface in the System namespace to enable you to implement the dispose pattern in your applications.

The IDisposable interface defines a single parameterless method named Dispose. You should use the Dispose method to release all of the unmanaged resources that your object consumed. If the object is part of an inheritance hierarchy, the Dispose method can also release resources that the base types consumed by calling the Dispose method on the parent type.

Invoking the Dispose method does not destroy an object. The object remains in memory until the final reference to the object is removed and the GC reclaims any remaining resources.

Many of the classes in the .NET Framework that wrap unmanaged resources, such as the StreamWriter class, implement the IDisposable interface. The StreamWriter class implements a TextWriter object for the purpose of writing text information to a stream. The stream could be a file, memory, or network stream. You should also implement the IDisposable interface when you create your own classes that reference unmanaged types.

Implementing the IDisposable Interface

To implement the IDisposable interface in your application, perform the following steps:

1. Ensure that the System namespace is in scope by adding the following using statement to the top of the code file.
   using System;
2. Implement the IDisposable interface in your class definition.
   ...

```
public class ManagedWord : IDisposable
{
  public void Dispose()
  {
    throw new NotImplementedException();
  }
}
```

3. Add a private field to the class, which you can use to track the disposal status of the object, and check whether the Dispose method has already been invoked and the resources released.

```
public class ManagedWord : IDisposable
{
  bool _isDisposed;
  ...
}
```

4. Add code to any public methods in your class to check whether the object has already been disposed of. If the object has been disposed of, you should throw an ObjectDisposedException.

```
public void OpenWordDocument(string filePath)
{
  if (this._isDisposed)
    throw new ObjectDisposedException("ManagedWord");
     ...
}
```

5. Add an overloaded implementation of the Dispose method that accepts a Boolean parameter. The overloaded Dispose method should dispose of both managed and unmanaged resources if it was called directly, in which case you pass a Boolean parameter with the value true. If you pass a Boolean parameter with the value of false, the Dispose method should only attempt to release unmanaged resources. You may want to do this if the object has already been disposed of or is about to be disposed of by the GC.

```
public class ManagedWord : IDisposable
{
  ...
   protected virtual void Dispose(bool isDisposing)
   {
      if (this._isDisposed)
         return;
      if (isDisposing)
      {
        // Release only managed resources.
        ...
      }
      // Always release unmanaged resources.
      ...
      // Indicate that the object has been disposed.
      this._isDisposed = true;
   }
}
```

6. Add code to the parameterless Dispose method to invoke the overloaded Dispose method and then call the GC.SuppressFinalize method. The GC.SuppressFinalize method instructs the GC that the resources that the object referenced have already been released and the GC does not need to waste time running the finalization code.

```
public void Dispose()
```

```
{
  Dispose(true);
  GC.SuppressFinalize(this);
}
```

After you have implemented the IDisposable interface in your class definitions, you can then invoke the Dispose method on your object to release any resources that the object has consumed. You can invoke the Dispose method from a destructor that is defined in the class.

Implementing a Destructor

You can add a destructor to a class to perform any additional application-specific cleanup that is necessary when your class is garbage collected. To define a destructor, you add a tilde (~) followed by the name of the class. You then enclose the destructor logic in braces.
The following code example shows the syntax for adding a destructor.

```
// Defining a Destructor
class ManagedWord
{
   ...
   // Destructor
   ~ManagedWord
   {
      // Destructor logic.
   }
}
```

When you declare a destructor, the compiler automatically converts it to an override of the Finalize method of the object class. However, you cannot explicitly override the Finalize method; you must declare a destructor and let the compiler perform the conversion.

If you want to guarantee that the Dispose method is always invoked, you can include it as part of the finalization process that the GC performs. To do this, you can add a call to the Dispose method in the destructor of the class.

The following code example shows how to invoke the Dispose method from a destructor.

```
// Calling the Dispose Method from a Destructor
class ManagedWord
{
   ...
   // Destructor
```

```
    ~ManagedWord
    {
        Dispose(false);
    }
}
```

Managing the Lifetime of an Object

Using types that implement the IDisposable interface is not sufficient to manage resources. You must also remember to invoke the Dispose method in your code when you have finished with the object. If you choose not to implement a destructor that invokes the Dispose method when the GC processes the object, you can do this in a number of other ways.

One approach is to explicitly invoke the Dispose method after any other code that uses the object. The following code example shows how you can invoke the Dispose method on an object that implements the IDisposable interface.

```
// Invoking the Dispose Method
var word = new ManagedWord();
 // Code to use the ManagedWord object.
word.Dispose();
```

Invoking the Dispose method explicitly after code that uses the object is perfectly acceptable, but if your code throws an exception before the call to the Dispose method, the Dispose method will never be invoked. A more reliable approach is to invoke the Dispose method in the finally block of a try/catch/finally or a try/finally statement. Any code in the scope of the finally block will always execute, regardless of any exceptions that might be thrown. Therefore, with this approach, you can always guarantee that your code will invoke the Dispose method.

The following code example shows how you can invoke the Dispose method in a finally block.

```
// Invoking the Dispose Method in a finally Block
var word = default(ManagedWord);
try
{
  word = new ManagedWord();
  // Code to use the ManagedWord object.
}
catch
{
   // Code to handle any errors.
}
```

```
finally
{
  if(word!=null)
    word.Dispose();
}
```

Note: When explicitly invoking the Dispose method, it is good practice to check whether the object is not null beforehand, because you cannot guarantee the state of the object.

Alternatively, you can use a using statement to implicitly invoke the Dispose method.
A using block is exception safe, which means that if the code in the block throws an exception, the runtime will still dispose of the objects that are specified in the using statement.

The following code example shows how to implicitly dispose of your object by using a using statement.

```
// Disposing Of an Object by Using a using Statement
using (var word = default(ManagedWord))
{
  // Code to use the ManagedWord object.
}
```

If your object does not implement the IDisposable interface, a try/finally block is an exception-safe approach to execute code to release resources. You should aim to use a try/finally block when it is not possible to use a using statement.

Module Six Assignment

Now that you have created some classes for your application, it's time to start thinking about your class heirarchies. In a typical development cycle, you would design your class heirarchies first, create your base classes, and then your sub classes. In a learning environment, you need to know how to create classes before you become inundated with inheritance heirarchies.

For this assignment:

1. Create a Person base class with common attributes for a person
2. Make your Student and Teacher classes inherit from the Person base class

3. Modify your Student and Teacher classes so that they inherit the common attributes from Person
4. Modify your Student and Teacher classes so they include characteristics specific to their type. For example, a Teacher object might have a GradeTest() method where a student might have a TakeTest() method.
5. Run the same code in Program.cs from Module 5 to create instances of your classes so that you can setup a single course that is part of a program and a degree path. Be sure to include at least one Teacher and an array of Students.
6. Ensure the Console.WriteLine statements you included in Homework 5, still output the correct information.
7. Share your code for feedback and ideas with your fellow students such as:

1. What other objects could benefit from inheritance in this code?
2. Can you think of a different hierarchy for the Person, Teacher, and Student? What is it?
3. Do **NOT** grade the answers to these two questions, they are merely for discussion and thought. Only grade the ability to implement inheritance in the code.

Challenge (Do NOT submit this to the peer review. This is for you use only)

- Create an instance of a Person object in code
- Create an instance of a Student object in code
- Assign the Student object to the Person object
- Access the properties of the Person instance you created
- What do you notice about the properties for the Person instance?
- Can you explain the behavior of the properties?