Module 11: Integrating with Unmanaged Code

Contents:

Module overview

Software systems can be complex and may involve applications that are implemented in a variety of technologies. For example, some applications may be managed Microsoft® .NET Framework applications, whereas others may be unmanaged C++ applications. You may want to use functionality from one application in another or use functionality that is exposed through Component Object Model (COM) assemblies, such as the Microsoft Word 14.0 Object Library assembly, in your applications.

In this module, you will learn how to interoperate unmanaged code in your applications and how to ensure that your code releases any unmanaged resources.

Objectives

After completing this module, you will be able to:

- Integrate unmanaged code into a Microsoft Visual C#® application by using the Dynamic Language Runtime (DLR).

- Control the lifetime of unmanaged resources and ensure that your application releases resources.

Lesson 1: Creating and Using Dynamic Objects

There are many different programming languages available, each with its own set of advantages and disadvantages and scenarios where it is better suited. Having the ability to consume components that are implemented in other languages enables you to reuse functionality that may already exist in your organization.

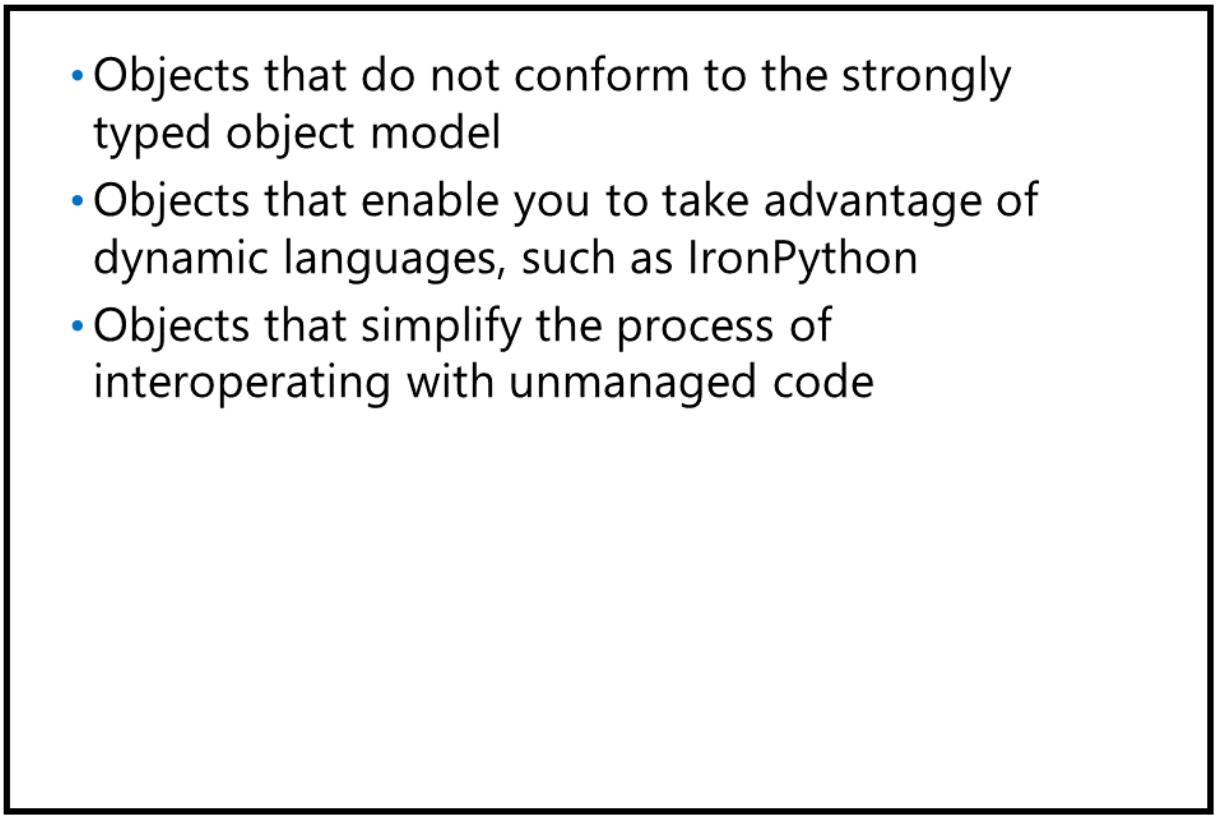
In this lesson, you will learn how to use the DLR to interoperate with unmanaged code.

Lesson objectives

After completing this lesson, you will be able to:

- Describe the purpose of dynamic objects.
- Describe the purpose of the DLR.
- Create a dynamic object.
- Invoke methods on a dynamic object.

What Are Dynamic Objects?



Visual C# is a strongly typed static language. When you create a variable, you specify the type of that variable, and you can only invoke methods and access members that this type defines. If you try to call a method that the type does not implement, your code will not compile. This is good because the compile time checks catch a large number of possible errors even before you run your code.

The .NET Framework provides dynamic objects so that you can define objects that are not constrained by the static type system in Visual C#. Instead, dynamic objects are not type checked until run time, which enables you to write code quickly without worrying about defining every member until you need to call them in your code.

You can use dynamic objects in your .NET Framework applications to take advantage of dynamic languages and unmanaged code.

Dynamic Languages

Languages such as IronPython and IronRuby are dynamic and enable you to write code that is not compiled until run time. Dynamic languages provide the following benefits:

- They enable faster development cycles because they do not require a build or

compile process.

- They have increased flexibility over static languages because there are no static types to worry about.
- They do not have a strongly typed object model to learn.

Dynamic languages do suffer from slower execution times in comparison with compiled languages, because any optimization steps that a compiler may take when compiling the code are left out of the build process.

Unmanaged Code

Visual C# is a managed language, which means that the Visual C# code you write is executed by a runtime environment known as the Common Language Runtime (CLR). The CLR provides other benefits, such as memory management, Code Access Security (CAS), and exception handling. Unmanaged code such as C++ executes outside the CLR and in general benefits from faster execution times and being extremely flexible.

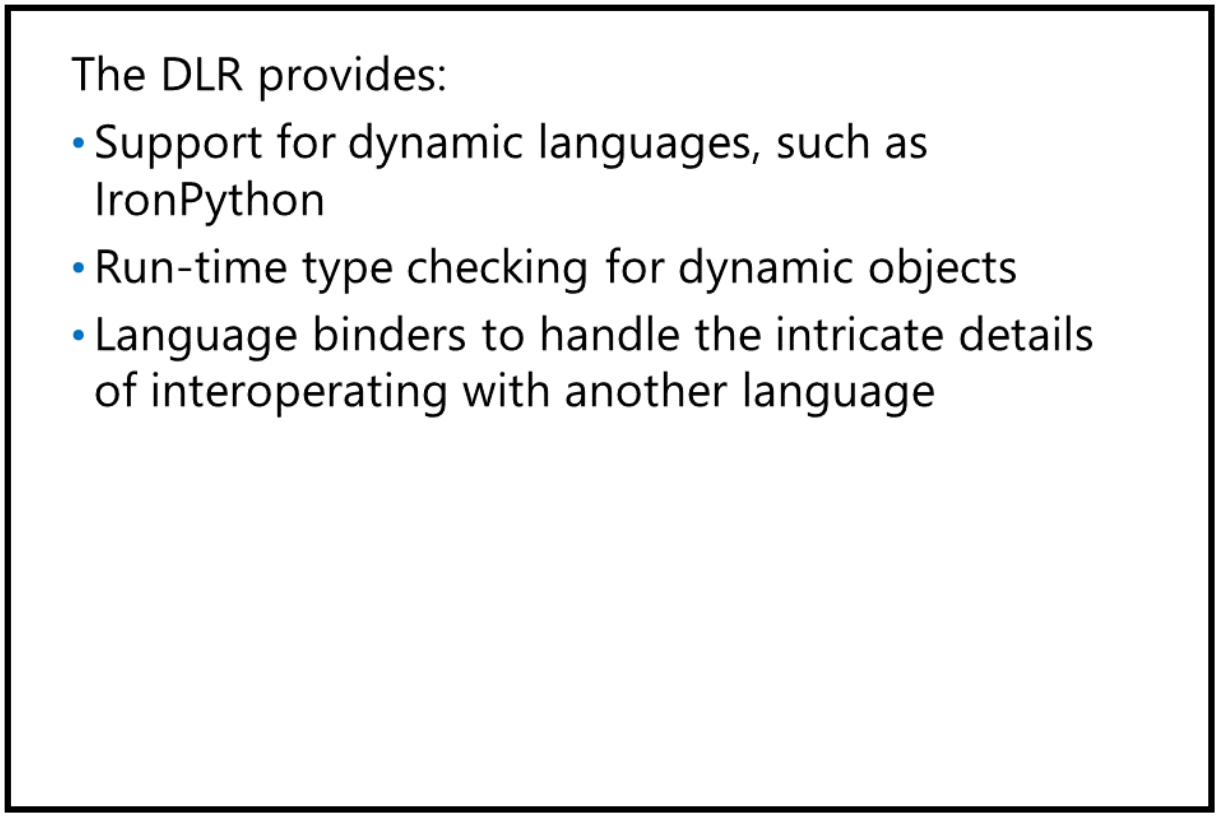
When you build applications by using the .NET Framework, it is a common use case to want to reuse unmanaged code. Maybe you have a legacy C++ system that was implemented a decade ago, or maybe you just want to use a function in the user32.dll assembly that Windows provides. The process of reusing functionality that is implemented in technologies other than the .NET Framework is known as interoperability. These technologies can include:

- COM
- C++
- Microsoft ActiveX®
- Microsoft Win32 application programming interface (API)

Dynamic objects simplify the code that is required to interoperate with unmanaged code.

> **Additional Reading:** For more information about dynamic objects, refer to the DynamicObject Class page at **https://aka.ms/moc-20483c-m11-pg1**.

What Is the Dynamic Language Runtime?



The .NET Framework provides the DLR, which contains the necessary services and components to support dynamic languages and provides an approach for interoperating with unmanaged components. The DLR is responsible for managing the interactions between the executing assembly and the dynamic object, for example, an object that is implemented in IronPython or in a COM assembly.

The DLR simplifies interoperating with dynamic objects in the followings ways:

- It defers type-safety checking for unmanaged resources until run time. Visual C# is a type-safe language and, by default, the Visual C# compiler performs type-safety checking at compile time.

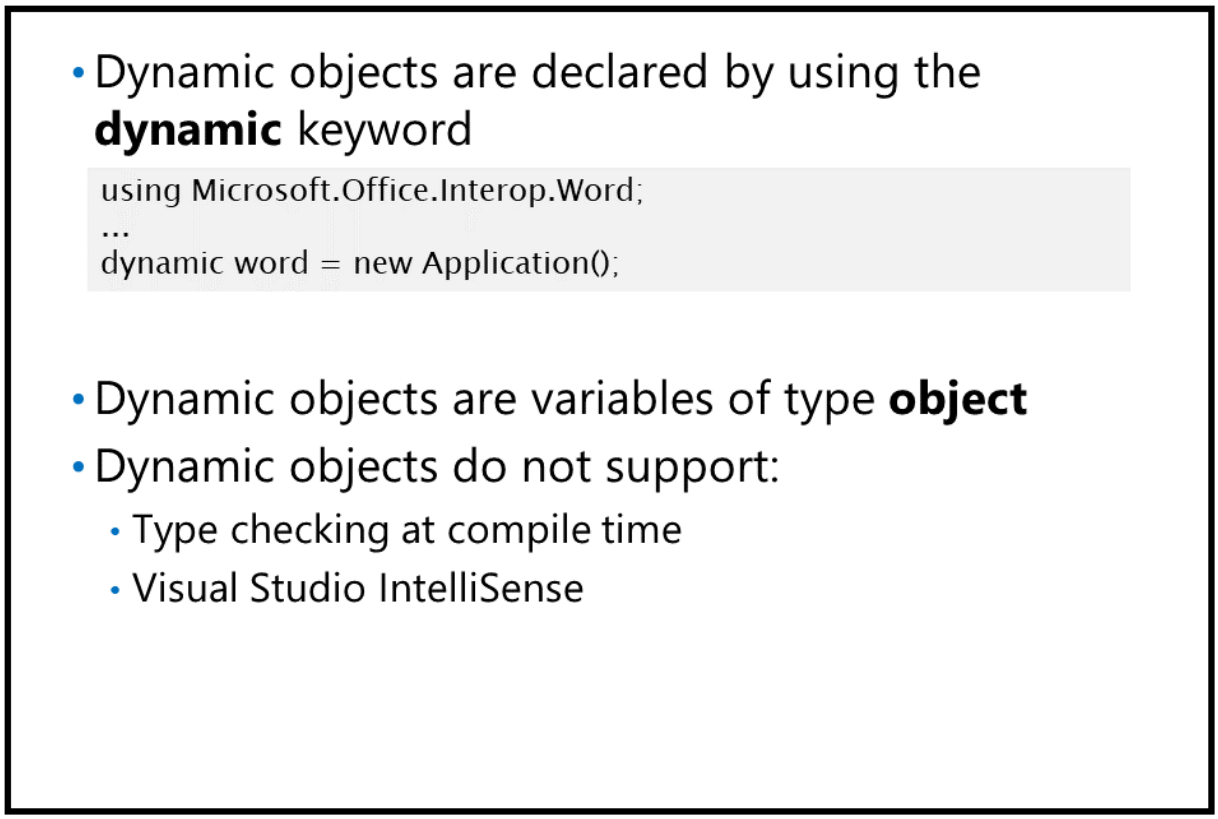
> **Note:** Type safety relies on the compiler being able to compile or interpret each of the components that the solution references. When interoperating with unmanaged components, the compile-time type-safety check is not always possible.

- It abstracts the intricate details of interoperating with unmanaged components, including marshaling data between the managed and unmanaged environments.

The DLR does not provide functionality that is pertinent to a specific language but provides a set of language binders. A language binder contains instructions on how to invoke methods and marshal data between the unmanaged language, such as IronPython, and the .NET Framework. The language binders also perform run-time type checks, which include checking that a method with a given signature actually exists in the object.

> **Additional Reading:** For more information about the DLR, see the Dynamic Language Runtime Overview page at **http://go.microsoft.com/fwlink/?LinkID=267858**.

Creating a Dynamic Object



The DLR infrastructure provides the **dynamic** keyword to enable you to define dynamic objects in your applications. When you use the **dynamic** keyword to declare a dynamic object, it has the following implications:

- It defines a variable of type **object**. You can assign any value to this variable and attempt to call any methods. At run time, the DLR will use the language binders to type check your dynamic code and ensure that the member you are trying to invoke exists.
- It instructs the compiler not to perform type checking on any dynamic code.
- It suppresses the Microsoft IntelliSense® feature because IntelliSense is unable to provide syntax assistance for dynamic objects.

To create a dynamic object to consume the Microsoft.Office.Interop.Word COM assembly, perform the following steps:

1. In your .NET Framework project, add a reference to the Microsoft.Office.Interop.Word COM assembly.
2. Bring the **Microsoft.Office.Interop.Word** namespace into scope.

The following code example shows how to create an instance of the **Application** COM class in the **Microsoft.Office.Interop.Word** namespace by using the **dynamic** keyword.

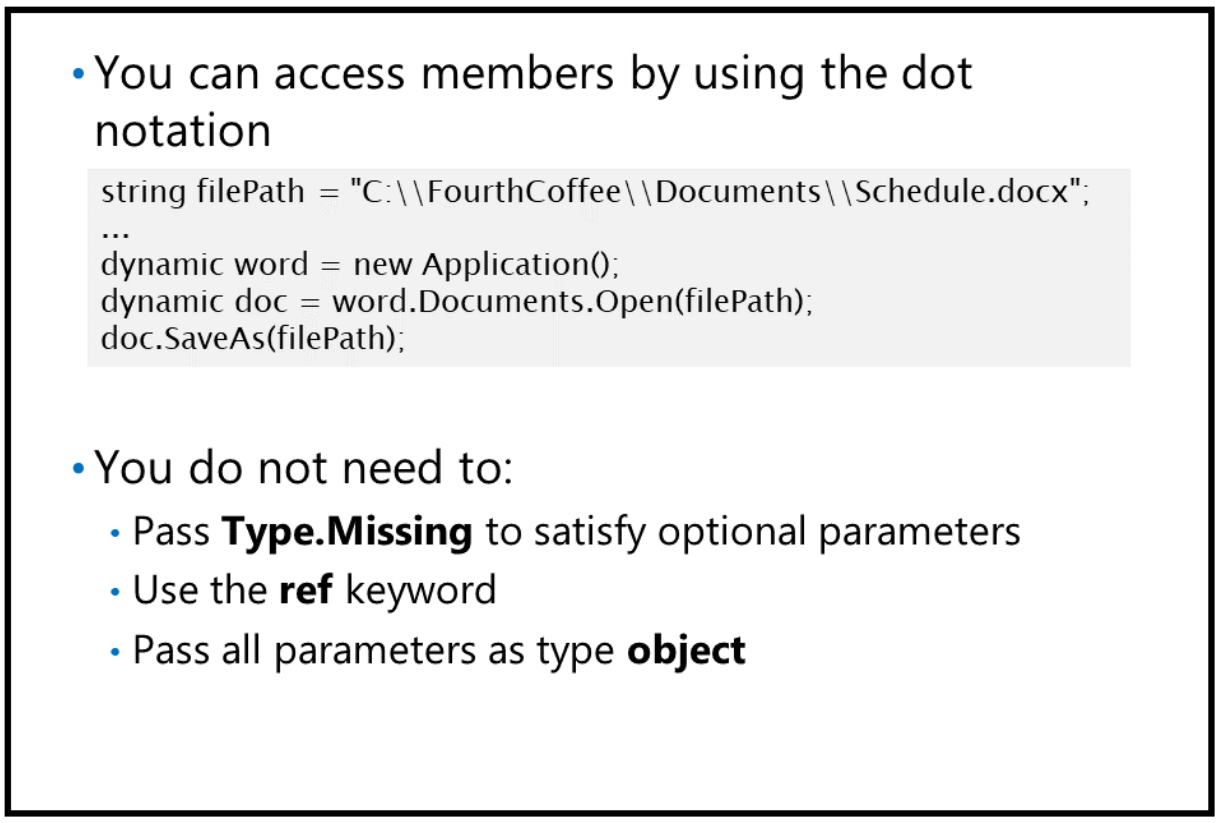
Dynamic Variable Declaration

```
using Microsoft.Office.Interop.Word;
...
dynamic word = new Application();
```

After you have created an instance of the class, you can use the members that it provides in the same way you would with any.NET Framework class.

> **Additional Reading:** For more information about the **dynamic** keyword, refer to the Using Type dynamic (C# Programming Guide) page at **https://aka.ms/moc-20483c-m11-pg2**

Invoking Methods on a Dynamic Object



After you have instantiated a dynamic object by using the **dynamic** keyword, you can access the properties and methods by using the standard Visual C# dot notation.

The following code example shows how you can use the **Add** method of the **Document** class in the Word interop library to create a new Word document.

Invoking a Method on a Dynamic Object

```
// Start Microsoft Word.
dynamic word = new Application();
...
// Create a new document.
dynamic doc = word.Documents.Add();
doc.Activate();
```

You can also pass parameters to dynamic object method calls as you would with any .NET Framework object.

The following code example shows how you can pass a string variable to a method that a dynamic object exposes.

Passing Parameters to a Method

```
string filePath = "C:\\FourthCoffee\\Documents\\Schedule.docx";
...
dynamic word = new Application();
dynamic doc = word.Documents.Open(filePath);
doc.SaveAs(filePath);
```

Traditionally when consuming a COM assembly, the .NET Framework required you to use the **ref** keyword and pass a **Type.Missing** object to satisfy any optional parameters that a method contains. Also, any parameters that you pass must be of type **object**. This can result in simple method calls requiring a long list of **Type.Missing** objects and code that is verbose and difficult to read. By using the **dynamic** keyword, you can invoke the same methods but with less code.

> **Best Practice:** When using dynamic objects in your code to encapsulate classes that are exposed in COM assemblies, use the Object Browser feature in Microsoft Visual Studio® to view the COM API.

> **Additional Reading:** For more information about how to create and consume dynamic methods, see the How to: Define and Execute Dynamic Methods page at **http://go.microsoft.com/fwlink/?LinkID=267860**.

Demonstration: Interoperating with Microsoft Word

In this demonstration, you will use dynamic objects to consume the Microsoft.Office.Interop.Word COM assembly in an existing .NET Framework application.

The application will use the Word object model to combine several text files that contain exception information into an exception report in Word format. The code uses

the Word object model to add data from the text files, line breaks, and formatting to generate the final document.

Demonstration steps

You will find the steps in the **Interoperating with Microsoft Word** section on the following page: https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD11_DEMO.md.

Lesson 2: Managing the Lifetime of Objects and Controlling Unmanaged Resources

When interoperating with unmanaged resources, it is important that you manage and release these resources when you are no longer using them.

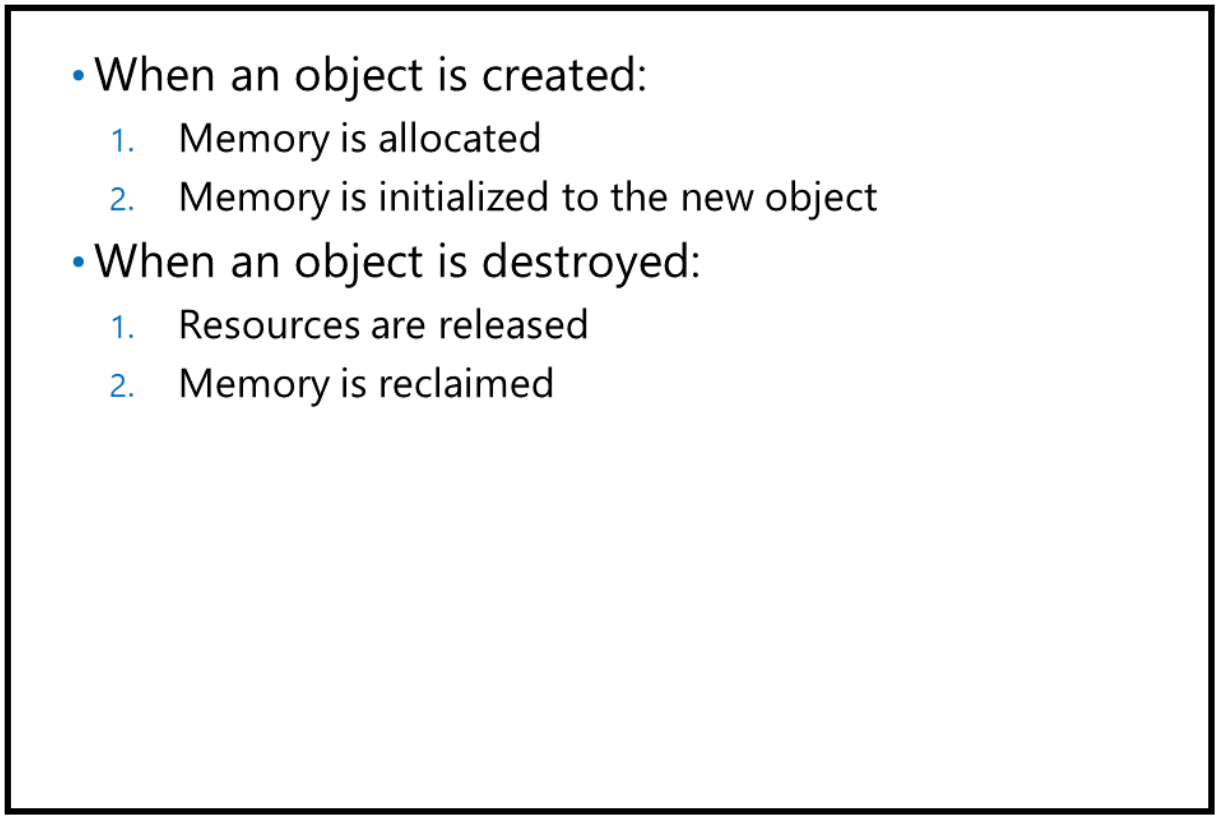
In this lesson, you will learn about the life cycle of a typical .NET Framework object and how to ensure that objects release the resources that they have been using when they are destroyed.

Lesson objectives

After completing this lesson, you will be able to:

- Describe the life cycle of a .NET Framework object.
- Implement the dispose pattern.
- Manage the lifetime of an object.

The Object Life Cycle



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 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.

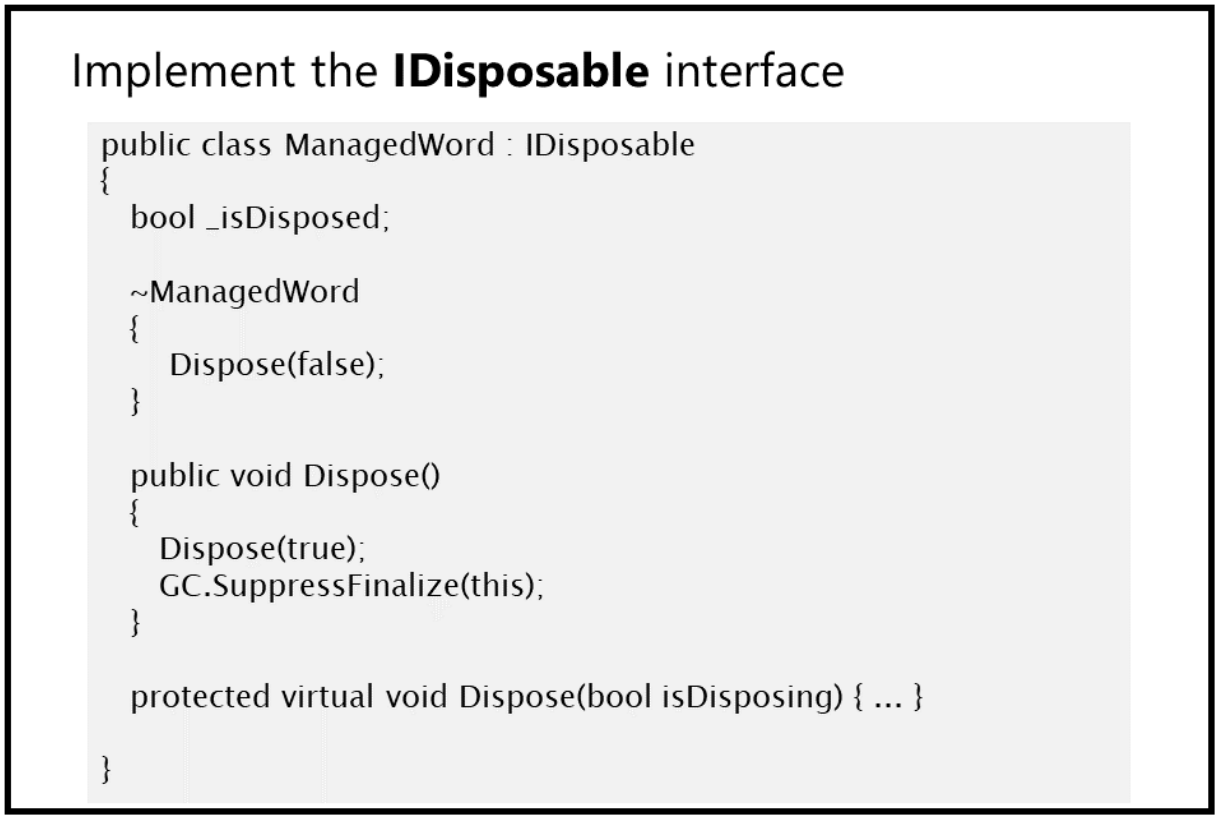
When you have finished with an object, you can dispose of it to release any resources, such as database connections and file handles, that it consumed. When you dispose of an object, the CLR uses a feature called the garbage collector (GC) to perform the following steps:

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.

> **Additional Reading:** For more information about GC, refer to the Garbage Collection page at **http://go.microsoft.com/fwlink/?LinkID=267861**.

Implementing the Dispose Pattern



The dispose pattern is a design pattern that frees resources that an object has used. The .NET Framework provides the **IDisposable** 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. You should also implement the **IDisposable** interface when you create your own classes that reference unmanaged types.

Implementing the IDisposable Pattern

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

1. Ensure that that the **System** namespace is in scope.

```
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.

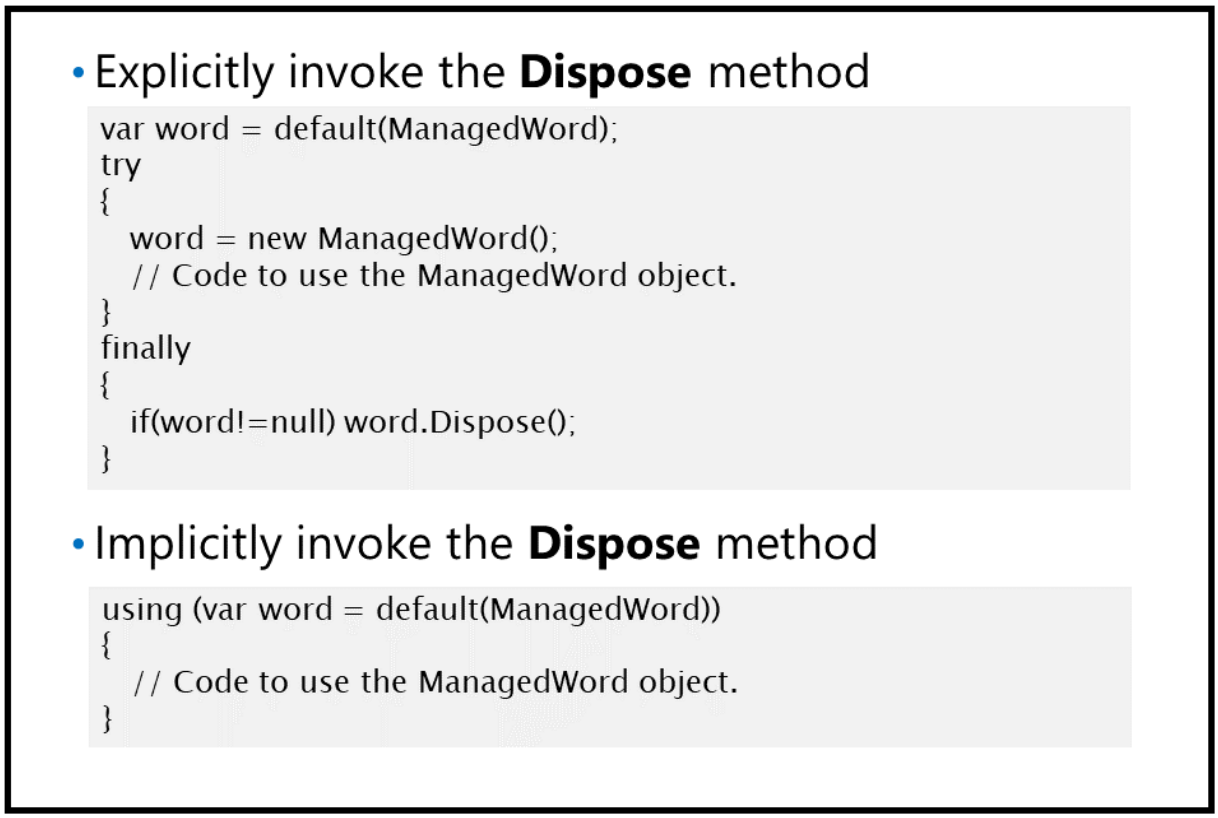
The following code example shows how to invoke the **Dispose** method from a destructor.

Calling the Dispose Method from a Destructor

```
class ManagedWord
{
    ...
    // Destructor
    ~ManagedWord
    {
        Dispose(false);
    }
}
```

Additional Reading: For more information about the **IDisposable** interface, refer to the IDisposable Interface page at **https://aka.ms/moc-20483c-m11-pg3**

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
{
    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.

> **Additional Reading:** For more information about **using** statements, refer to the using Statement (C# Reference) page at **https://aka.ms/moc-20483c-m11-pg4**.

Demonstration: Upgrading the Grades Report Lab

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: Upgrading the Grades Report Lab** section on the following page: https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD11_DEMO.md.

Lab: Upgrading the Grades Report

Scenario

You have been asked to upgrade the grades report functionality to generate reports in Word format. In Module 6, you wrote code that generates reports as an XML file; now you will update the code to generate the report as a Word document.

Objectives

After completing this lab, you will be able to:

- Use dynamic types.
- Manage object lifetime and resources.

Lab setup

Estimated Time: 60 minutes

You will find the high-level steps on the following page: **https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD11_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_MOD11_LAK.md**.

Exercise 1: Generating the Grades Report by Using Word

Scenario

In this exercise, you will update the reporting functionality to generate reports in Word format.

First you will review the existing code in the **WordWrapper** class that appends headings, text, breaks, and carriage returns to a document. You will write code to create an instance of Word, create a blank Word document, and save a Word document. You will then review the code in the **GenerateStudentReport** method to create a blank document, add a heading and grade data to the document, and save the document using the methods that you reviewed and wrote in the **WordWrapper** class. You will run this method as a separate task. Finally, you will build and test the application and verify that the reports are now generated in Word format.

Result: After completing this exercise, the application will generate grade reports in Word format.

Exercise 2: Controlling the Lifetime of Word Objects by Implementing the Dispose Pattern

Scenario

In this exercise, you will write code to ensure that Word is correctly terminated after generating a grades report.

You will begin by observing that currently the Word object remains in memory after the application has generated a report. You will verify this by observing the running tasks in Task Manager. You will update the code in the **WordWrapper** class to implement the dispose pattern to ensure correct termination of the Word instance. You will then update the code in the **GenerateStudentReport** method to ensure that the **WordWrapper** object is disposed of when the method finishes. Finally, you will build and test the application and verify that Word now closes after the report is generated.

Result: After completing this exercise, the application will terminate Word correctly after it has generated a grades report.

Module review and takeaways

In this module, you have learned how to interoperate with COM assemblies and how to ensure that your objects dispose of any resources they consume.

Review Question(s)

Check Your Knowledge

Select the best answer

Which of the following statements best describes the dynamic keyword?

It defines an object of type object and instructs the compiler to perform type checking.

It defines a nullable object and instructs the compiler to defer type checking.

It defines an object of type object and instructs the compiler to defer type checking.

It defines a nullable object and instructs the compiler to perform type checking.

Check answer Show solution Reset

Check Your Knowledge

True or False Question

You can use a using statement to implicitly invoke the Dispose method on an object that implements the IDisposable pattern.

True False

Check answer Reset