Microsoft® Official Course

Module 11

Integrating with Unmanaged Code

Microsoft®

Module Overview

- Creating and Using Dynamic Objects
- Managing the Lifetime of Objects and Controlling Unmanaged Resources

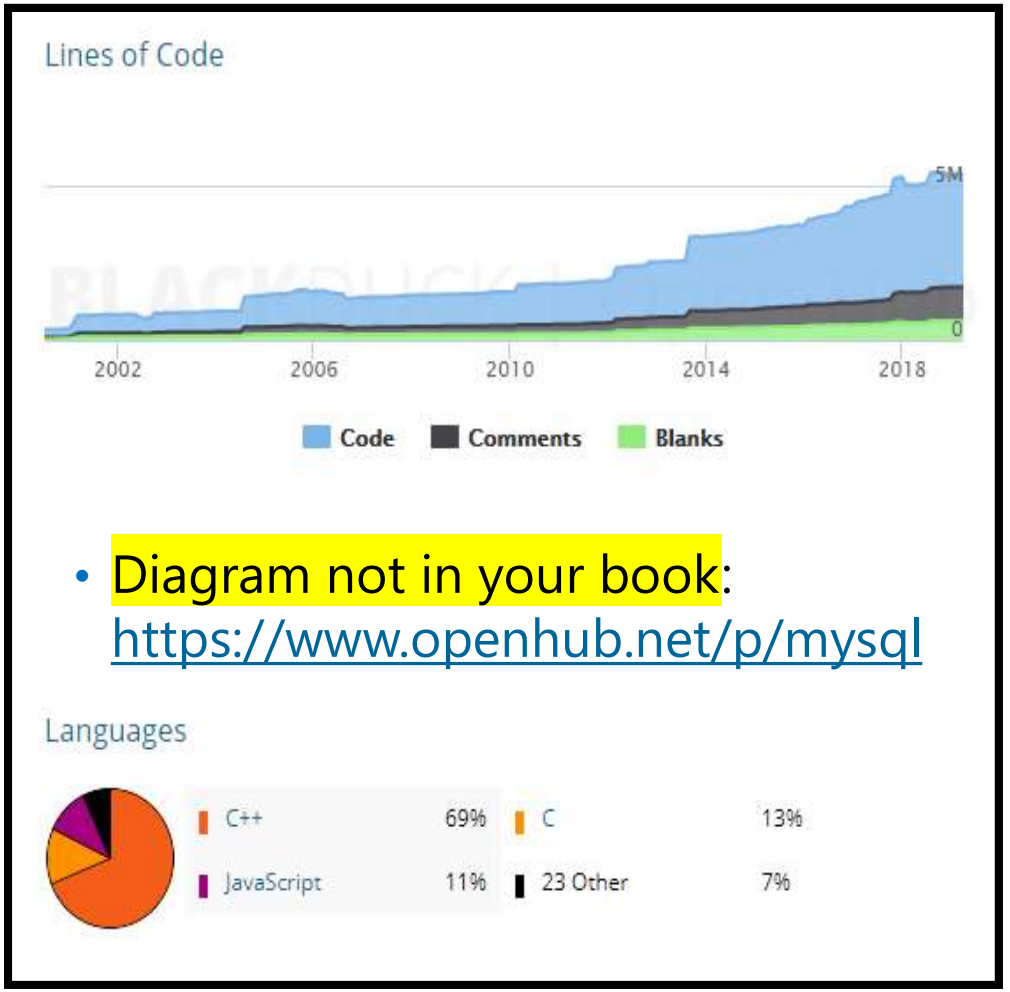
- **Note**: The samples, demos, and the lab in this module focus on interoperating with Microsoft® Word, but the concept would be the same with other Component Object Model (COM) applications.

Lesson 1: Creating and Using Dynamic Objects

- What Are Dynamic Objects?
- What Is the Dynamic Language Runtime?
- Creating a Dynamic Object
- Invoking Methods on a Dynamic Object
- Demonstration: Interoperating with Microsoft Word

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.
- Having the ability to consume components that are implemented in other languages enables you to reuse existing functionality



- Diagram not in your book: https://www.openhub.net/p/mysql

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 way the compile time checks can catch a large number of possible errors even before you run your code.

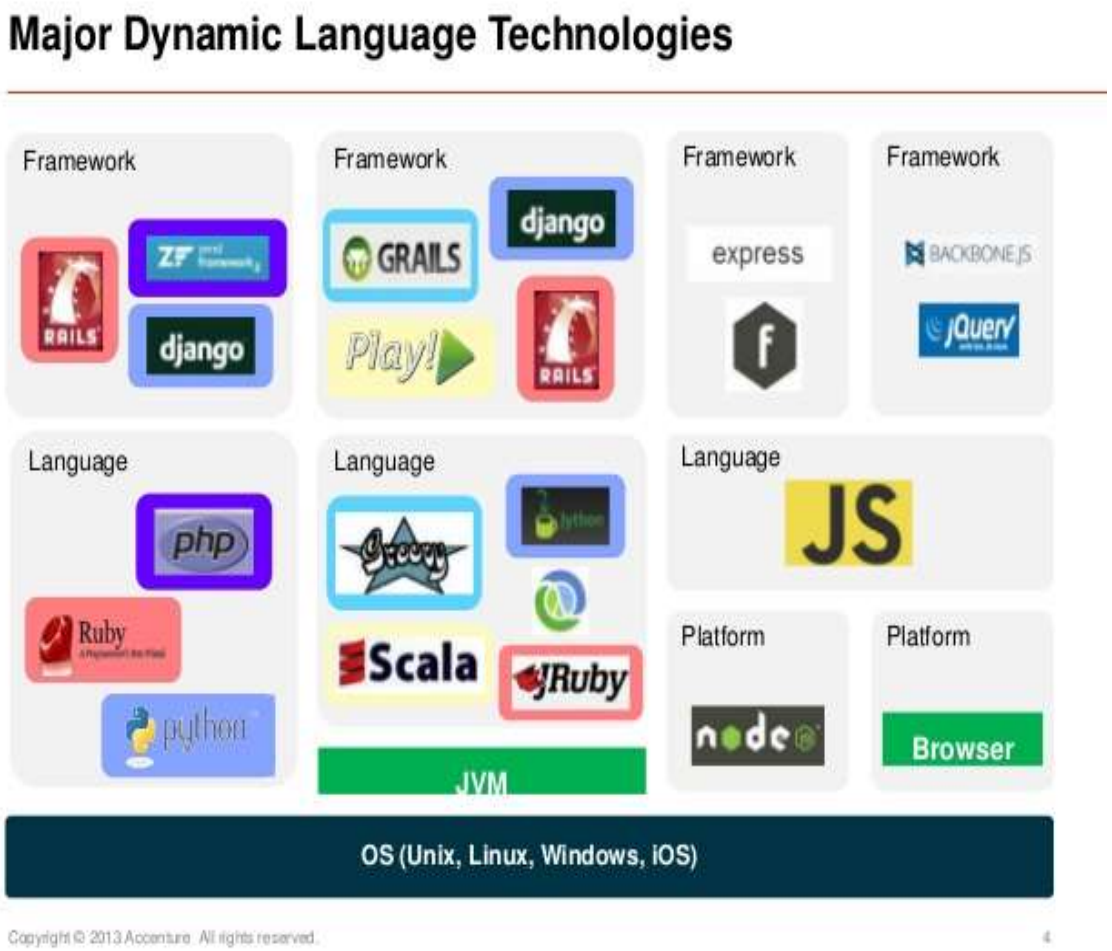
- The .NET Framework provides dynamic objects so that you can define objects that are not constrained by the static type system in Visual C#. What are they?
  - Objects that do not conform to the strongly typed object model
  - Objects that enable you to take advantage of dynamic languages, such as IronPython
  - Objects that simplify the process of interoperating with unmanaged code

- Dynamic objects enable you to write code in your .NET Framework applications by using languages other than Visual C#; this means that you can write code that does not conform to the strongly typed Visual C# object model.
  - The focus of this lesson is interoperating with unmanaged code by using dynamic objects, not implementing logic by using dynamic languages.

Dynamic Languages

- Dynamic languages (such as **IronPython** and **IronRuby**) enable you to write code that is not compiled until run time. Benefits:
  - faster development cycles because they do not require a build or compile process.
  - increased flexibility over static languages because there are no static types to worry about.
  - do not have a strongly typed object model to learn.
- A drawback:
  - Dynamic languages are typically slower (compiled languages may optimize the code …)

source: google images



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.

- 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.
  - https://docs.microsoft.com/en-us/dotnet/api/system.dynamic.dynamicobject?redirectedfrom=MSDN&view=netframework-4.8

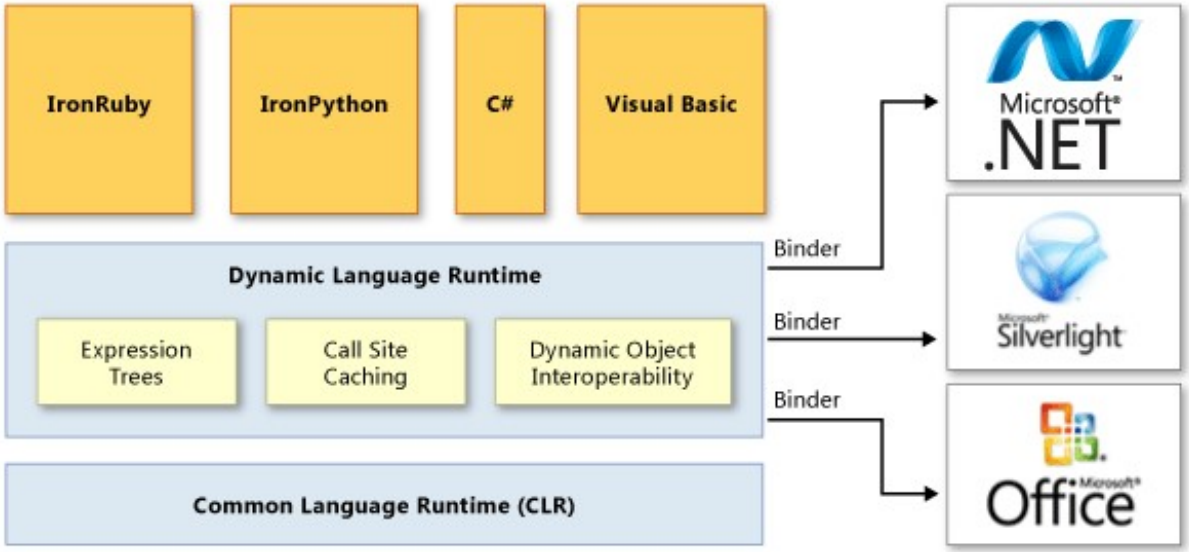
What Is the Dynamic Language Runtime (DLR)?

The .NET Framework provides the DLR, which:

- Support for dynamic languages, such as IronPython
- Run-time type checking for dynamic object
  - 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.
- Language binders to handle the intricate details of interoperating with another language
  - 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 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.

See also: https://docs.microsoft.com/en-us/dotnet/framework/reflection-and-codedom/dynamic-language-runtime-overview

DLR architecture



Creating a Dynamic Object

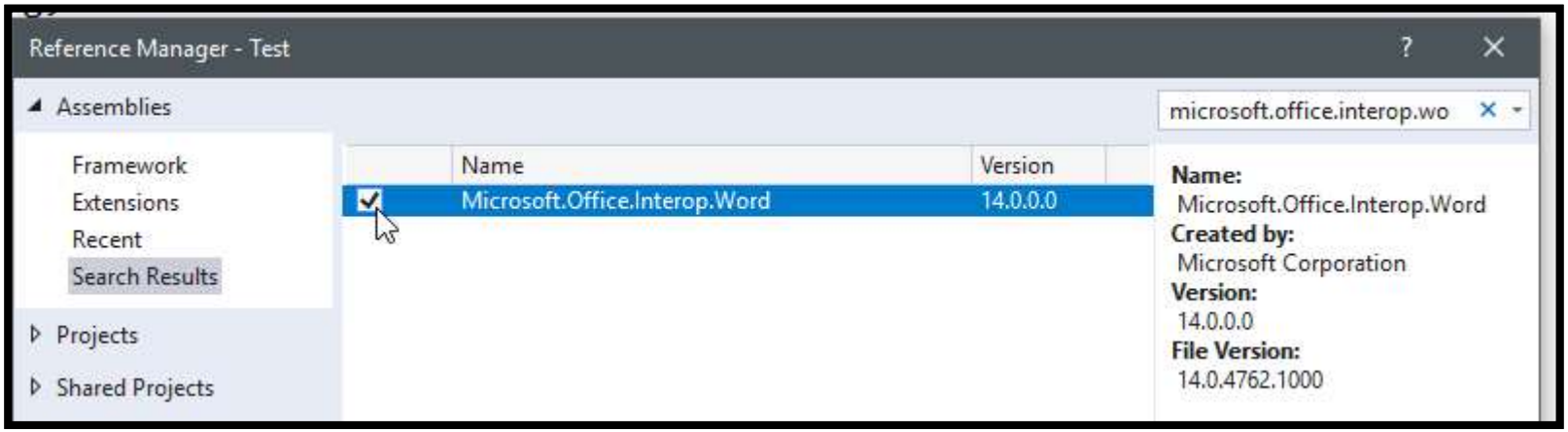
- Dynamic objects are declared by using the **dynamic** keyword
  - Dynamic objects are variables of type **object**
    - You can assign any value to this variable and attempt to call any methods. At run time, the DLR will use the language binders to type check your dynamic code and ensure that the member you are trying to invoke exists.
  - It instructs the compiler not to perform type checking on any dynamic code.
    - The type is a static type, but an object of type dynamic bypasses static type checking.
  - IntelliSense is unable to provide syntax assistance for dynamic objects
    - Note: you should not use **dynamic** objects as a replacement for the **var** keyword or the type name, because of the lack of compile-time type checking.

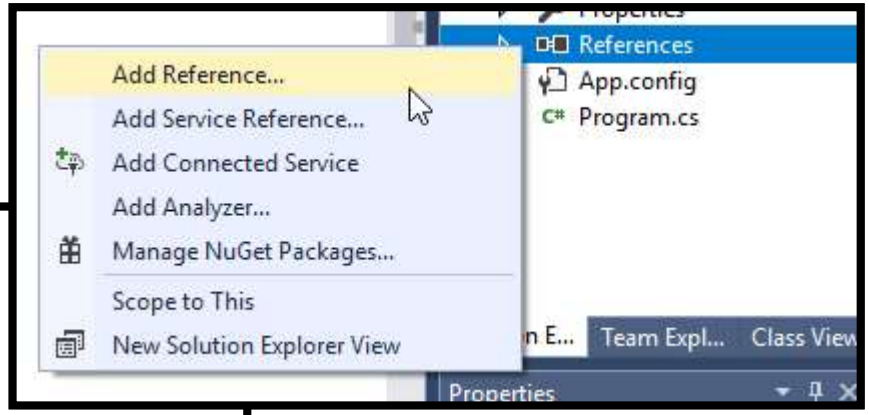
- To create a dynamic object to consume the **Microsoft.Office.Interop.Word** COM assembly, perform the following steps:
  - In your .NET Framework project, add a reference to the **Microsoft.Office.Interop.Word** COM assembly.
  - Bring the **Microsoft.Office.Interop.Word** namespace into scope.

```
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

Creating a Dynamic Object






```
using Microsoft.Office.Interop.Word;

namespace linkedlist
{
    class list
    {

        static void Main(string[] args)
        {
            dynamic word = new Application();

        }
    }
}
```

interface Microsoft.Office.Interop.Word.Application
Represents the Microsoft Office Word application.

Invoking Methods on a Dynamic Object

You can access members by using the **dot notation**

- pass parameters to dynamic object method calls as you would with any .NET Framework object

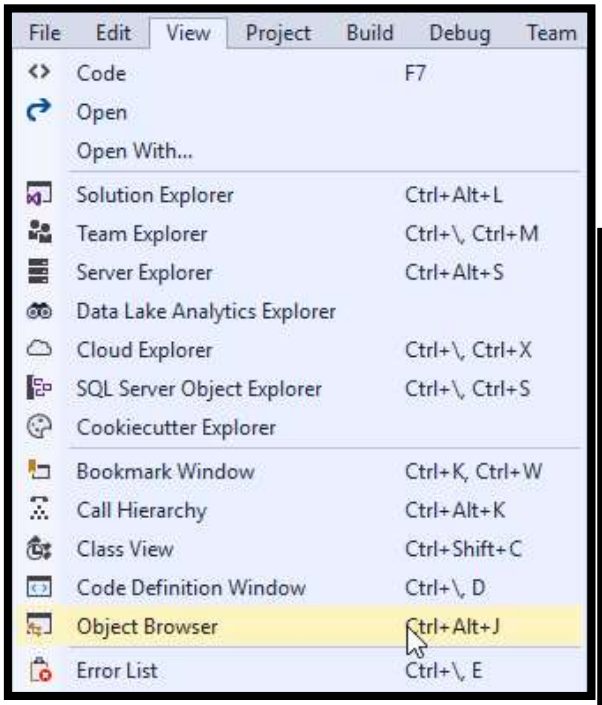
```
string filePath = "W:\\Schedule.docx";

// Start Microsoft Word
dynamic word = new Application();

// Create a new document.
dynamic doc = word.Documents.Open(filePath);

...
doc.SaveAs(filePath);
```

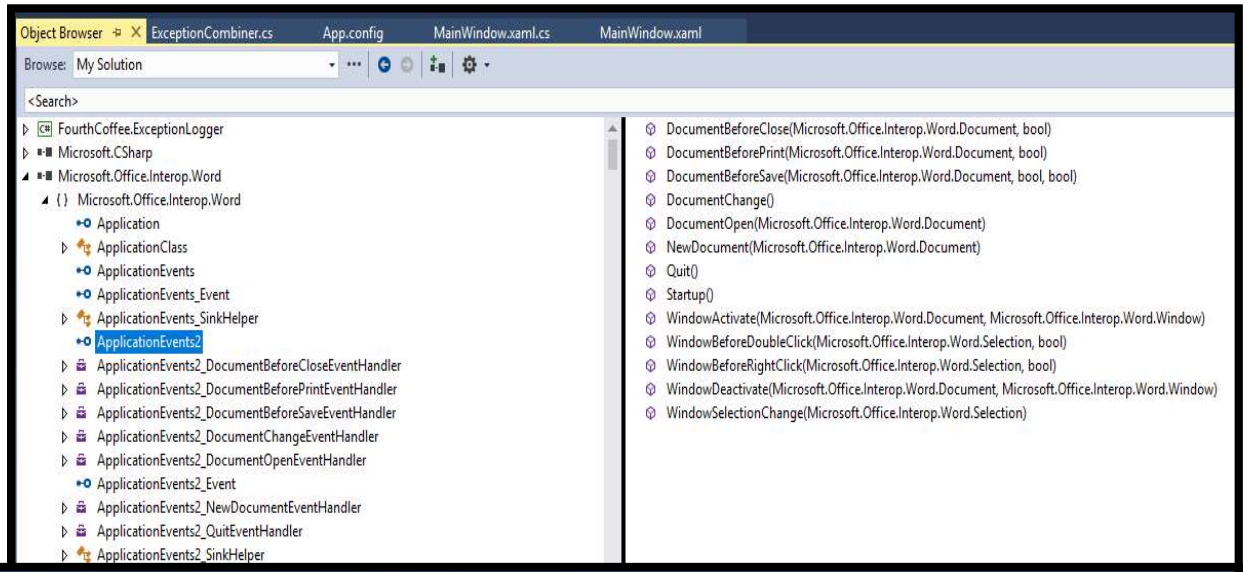
You do not need to (typically needed when consuming COM assemblies):

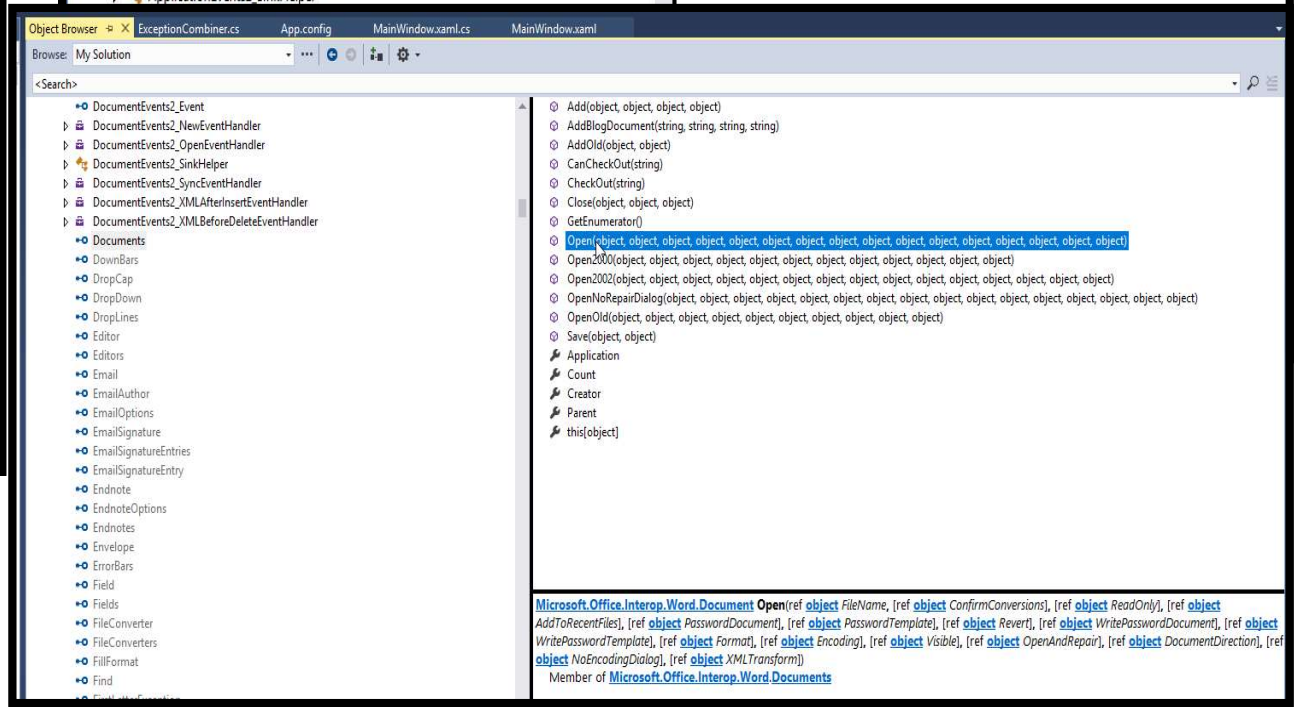
- Pass **Type.Missing** to satisfy optional parameters
- Use the **ref** keyword
- Pass all parameters as type **object**

View the API in the **Object Browser** window.

- View the **Microsoft.Office.Interop.Word** application programming interface (API) in the **Object Browser** window.







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

- The Object Life Cycle
- Implementing the Dispose Pattern
- Managing the Lifetime of an Object
- Demonstration: Upgrading the Grades Report Lab

The Object Life Cycle

- 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. Memory is allocated
     - 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.
  2. Memory is initialized to the new object
     - you can control the initialization of an object by implementing a constructor.

- When an object is destroyed:
  1. Resources are released
     - E.g. database connections and file handles, that it consumed
  2. Memory is reclaimed
     - Note: The **Garbage Collector** (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.

The Object Life Cycle – extra … not in your book

- Let's create a student class with both a constructor and a destructor (also called finalizer in C#). Let's see how this works with objects and the new operator.
  - If time, write examples of inheritance and see how the constructors/destructors are called
  - A finalizer will impact the performance of your type so don't implement it unless you need it.

- Finalizers (destructors) are used to perform any necessary final clean-up when a class instance is being collected by the garbage collector.
  - See more in here … https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/destructors

- For unmanaged resources, you need allocate/deallocate the unmanaged memory (for the managed resources, Garbage collector will take care of that)
  - See next slide …
  - Also see: https://docs.microsoft.com/en-us/dotnet/standard/garbage-collection/implementing-dispose

Remarks

- Finalizers cannot be defined in structs. They are only used with classes.
- A class can only have one finalizer.
- Finalizers cannot be inherited or overloaded.
- Finalizers cannot be called. They are invoked automatically.
- A finalizer does not take modifiers or have parameters.

Implementing the Dispose Pattern

implement the **IDisposable** interface (the dispose pattern) when you create your own classes that reference unmanaged types:

- It defines a single **parameterless** method named **Dispose**.
- Add a private field **_isDisposed** - tracks the disposal status of the object, and check whether the **Dispose** method has already been invoked and the resources released.
- use the **Dispose** method to release all of the unmanaged resources that your object consumed.
- 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**.
- Add an overloaded implementation of the Dispose method that accepts a Boolean parameter. It should dispose of both managed and unmanaged resources if it was called directly(parameter with the value true). If you pass a Boolean parameter **false**, the Dispose method should only attempt to release unmanaged resources.
- Add code to the parameterless Dispose method to invoke the overloaded Dispose method and then call the **GC.SuppressFinalize** to instruct 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 class ManagedWord : IDisposable
{
   bool _isDisposed;

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

   public void Dispose()
   {
      Dispose(true);
      GC.SuppressFinalize(this);
   }

   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;
   }

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

Implementing the Dispose Pattern – example

- for a simple class with no unmanaged resources and a collection of IDisposable objects, your class might look something like this:

- Source: https://theburningmonk.com/2009/12/the-c-dispose-pattern/

```
public sealed class MyClass : IDisposable
{
      IList<MyObject> objects;  // MyClass holds a list of objects
      private bool _disposed;   // boolean flag to stop us calling Dispose(twice)

      public void Dispose()
      {
            Dispose(true);
            GC.SuppressFinalize(this);
      }

      private void Dispose(bool disposing)
      {
            if (!_disposed)
            {
                  // call Dispose on each item in the list
                  if (disposing)
                  {
                        foreach (var o in objects)
                        {
                              // check if MyObject implements IDisposable
                              var d = o as IDisposable();
                              if (d != null) d.Dispose();
                        }
                  }
            _disposed = true;
            }
      }
}
```

Managing the Lifetime of an Object

- Explicitly invoke the **Dispose** method
  - after code that uses the object
    OR
  - in a **try/catch/finally** or **try/finally** block.

```
var word = default(ManagedWord);
try
{
   word = new ManagedWord();
   // Code to use the ManagedWord object.
}
finally
{
   if(word!=null) word.Dispose();
}
```

- Implicitly invoke the **Dispose** method
  - by using a **using** statement.

```
using (var word = default(ManagedWord))
{
   // Code to use the ManagedWord object.
}
```

    - The using statement ensures that Dispose is called even if an exception occurs within the using block.
    - You can achieve the same result by putting the object inside a try block and then calling Dispose in a finally block; in fact, this is how the using statement is translated by the compiler

- See also:
  - https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/using-statement

Module Review and Takeaways

- **Question:** Which of the following statements best describes the **dynamic** keyword?
  - ( )Option 1: It defines an object of type object and instructs the compiler to perform type checking.
  - ( )Option 2: It defines a nullable object and instructs the compiler to defer type checking.
  - ( )Option 3: It defines an object of type object and instructs the compiler to defer type checking.
  - ( )Option 4: It defines a nullable object and instructs the compiler to perform type checking.

- **Question:** You can use a **using** statement to implicitly invoke the **Dispose** method on an object that implements the **IDisposable** pattern.
  - ( )False
  - ( )True