CSC310-A

Module 5

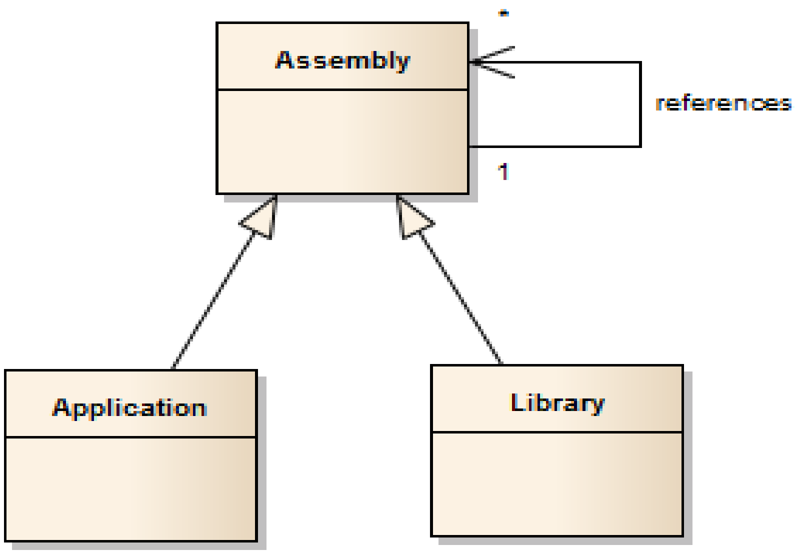
Advanced Object Access & Asynchronous Tasks

1 Assembly & Type

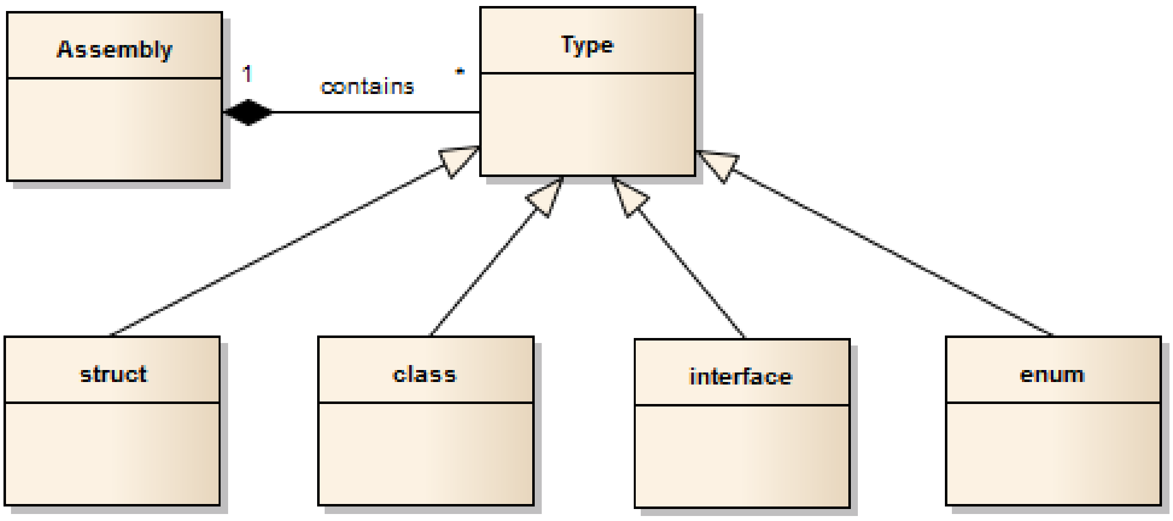
1.1 Assembly & Type

In .NET we build and use assemblies. You can build an application assembly packaged as an *executable program* or a library assembly packaged as a *dynamic-link library*. It is possible for one assembly to use another assembly by referencing it at compilation time. Only the **mscorlib** assembly is automatically referenced by .NET compilers. Any other assembly that you need to use has to be explicitly referenced.

Building and using assemblies



An assembly can contains types



Referencing assemblies does not mean that they will automatically be loaded just that the assembly will be deployed together with the application. An assembly is loaded if at least one type is used in your code. Call **GetAssemblies** method in **AppDomain** to find out which assemblies are currently loaded. Since this is something that may be useful for debugging, you can implement this code as a helper method.

Code to check assemblies currently loaded: Symbion\DebugHelper.cs

```
public static class DebugHelper {
    public static void ShowLoadedAssemblies() {
        AppDomain domain = AppDomain.CurrentDomain;
        Assembly[] assemblies = domain.GetAssemblies();
        foreach (Assembly assembly in assemblies)
            if (Debugger.IsAttached) Debug.WriteLine(assembly.FullName);
            else Console.WriteLine(assembly.FullName);
    }
}
```

There are additional methods in **Assembly** class to access loaded assemblies that is relative to the current code that is currently running.

Additional assembly methods

GetEntryAssembly	*Returns the application assembly*
GetExecutingAssembly	*Returns the assembly containing the currently running method*
GetCallingAssembly	*Returns the assembly calling the current method*
GetAssembly	*Returns the assembly containing a specific type*

Accessing loaded assemblies

```
DebugHelper.ShowLoadedAssemblies();
Assembly a1 = Assembly.GetEntryAssembly();
Assembly a2 = Assembly.GetExecutingAssembly();
Assembly a3 = Assembly.GetCallingAssembly();
Assembly a4 = Assembly.GetAssembly(typeof(Program));
Assembly a5 = typeof(DebugHelper).Assembly;
Console.WriteLine(a1.FullName);
Console.WriteLine(a2.FullName);
Console.WriteLine(a3.FullName);
Console.WriteLine(a4.FullName);
Console.WriteLine(a5.FullName);
```

Assemblies can be dynamically loaded. Since assemblies can be loaded based on the identity of the assembly rather that it's physical location you need to understand what an assembly identity is.

1.2 Assembly Identity

Symbion library at the moment is a private assembly. This means that it can only be used by assemblies in the same directory. To create a shared assembly you must sign the assembly with a key file. Use *Signing* page in *Project Properties* window to sign the assembly with a new key file named as **Symbion**. If other users have access to the project or solution, you should assign a password to protect the file. This provides the assembly with an unique assembly identity. While *Name*, *Version* and *Culture* can be duplicated, the *PublicKeyToken* is unique to each key.

Identity of an assembly

```
Assembly Identity = Name + Version + Culture + PublicKeyToken
```

A shared assembly can then be deployed together with the application or placed into the *Global Assembly Cache* (GAC). The .NET Runtime will look in the current directory for an assembly before loading it from the GAC. You can use the **gacutil** SDK tool to manage assemblies in the GAC. Be careful when removing an assembly from the GAC as you may remove other assemblies that have the same name. You should use the full assembly identity when removing assemblies. Also note that Visual Studio will not deploy any assembly that is already in the GAC so do not install them into the GAC on your development machine.

Installing an assembly into the GAC

```
C:\CSDEV\SRC\Module4\Symbion\bin\Debug> gacutil /i Symbion.dll
```

Checking identity of assemblies in the GAC

```
C:\CSDEV\SRC\Module4\Symbion\bin\Debug> gacutil /l Symbion
```

Removing an assembly from the GAC

```
C:\CSDEV\SRC\Module4\Symbion\bin\Debug> gacutil /u Symbion, Version=...
```

Signing an assembly guarantees that the correct assembly will always be loaded base on the identity of the assembly that was referenced during compilation. No one would be able to replace your assembly with a fake version since it can never have the same identity unless signed with the same key. A signed assembly cannot be tampered with as a .NET assembly will refuse to load in an assembly that has been changed since it was signed. For security you should sign all your assemblies including the application. Note that unsigned assemblies cannot be used from signed assemblies as it will break the security so you may not be able to use some third-party non-open source libraries that are unsigned.

1.3 Loading Assemblies

You can dynamically load assemblies at runtime instead of referencing them. This will allow you to decide which libraries to load at runtime instead of at compilation time. It will allow you to load different libraries without having to change and re-compile your code. There are a multiple load methods available in **Assembly** class. Use the **Load** method to load an assembly either in the same directory or from the Global Assembly Cache (GAC). Use **LoadFile** method to load an assembly using an absolute file path. Use **LoadFrom** instead if you wish to support relative file paths as well. You can also use this method to load assemblies from web sites by using a URL instead of a file path if this operation is allowed by the current code access security policy.

Loading assemblies from different locations:

```
Assembly a6 = Assembly.Load("Symbion");
Assembly a7 = Assembly.LoadFile(@"C:\CSDEV\SRC\Module5\Symbion\bin\Debug\Symbion.dll");
Assembly a8 = Assembly.LoadFrom(@"C:\CSDEV\SRC\Module5\Symbion\bin\Debug\Symbion.dll");
Assembly a9 = Assembly.LoadFrom(@"Symbion.dll");
```

To load assemblies from the GAC, you need to provide the full assembly identity and not just the name. Full identity of an assembly includes name, version, culture code and the public key token as shown below. T You will be able to access the identity of a loaded assembly by calling **GetName** method to get **AssemblyName** that stores the identity of the assembly.

If you do not sign an assembly when you built it, it becomes a private assembly and cannot be placed into the GAC since it does not have a unique identity. By signing an assembly, it becomes a shared assembly. Shared assemblies can be placed into the GAC because it can have a unique identity. You can sign multiple assemblies with the same key as long as the rest of the identity is not same so that they do not overwrite each other in the GAC.

Loading an assembly from the GAC

```
string id = "PresentationCore, " +
    "Version=3.0.0.0, " +
    "Culture=neutral, " +
    "PublicKeyToken=31bf3856ad364e35";
Assembly a10 = Assembly.Load(id);
Console.WriteLine(a10.FullName);
```

Accessing assembly identity

```
AssemblyName assemblyName = a10.GetName();
Console.WriteLine(assemblyName.Name);
Console.WriteLine(assemblyName.Version);
Console.WriteLine(assemblyName.CultureInfo);
byte[] token = assemblyName.GetPublicKeyToken();
Console.WriteLine(BitConverter.ToString(token));
Console.WriteLine(BitConverter.ToString(token).Replace("-","").ToLower());
```

Regardless of how an assembly is loaded, you can then obtain all public types in it by using **GetExportedTypes** method. Since this is quite useful for debugging you can add a method in **DebugHelper** class to show all the types in a particular assembly.

Displaying all types in an assembly: Symbion\DebugHelper.cs

```
public static void ShowTypes(this Assembly assembly) {
    Type[] types = assembly.GetExportedTypes();
    Console.WriteLine("Types in assembly: {0}", assembly.FullName);
        foreach (Type type in types) Console.WriteLine("\t{0}", type.FullName);
}
```

If you need to retrieve a specific type, then just call the **GetType** method with a full type name. If the type does not exist, **null** is returned instead. Once you have a type you can access the type using reflection. Use **Activator** to create an instance from its type.

Locate a type and creating an object from the type

```
Type type = a9.GetType("Symbion.DebugLogger");
object obj = Activator.CreateInstance(type);
```

1.4 Assembly Attributes

Additional information can be attached to an assembly using .NET attributes. You can find this in **AssemblyInfo** source file in **Properties** folder. Most of the information is only for reference except for **AssemblyVersion** and **AssemblyCulture** that becomes part of the assembly identity. Use the following attributes for **Symbion** library.

Assembly version information: Symbion\Properties\AssemblyInfo.cs

```
using System.Reflection;
using System.Runtime.InteropServices;

[assembly: AssemblyTitle("Symbion")]
[assembly: AssemblyDescription("A basic smart client application framework")]
[assembly: AssemblyCopyright("Copyright (c) Symbolicon Systems 2011")]
[assembly: AssemblyProduct("Advanced Visual C#")]
[assembly: AssemblyCompany("Symbolicon Systems")]
[assembly: AssemblyFileVersion("1.0.0.0")]
[assembly: AssemblyVersion("1.0.0.0")]
[assembly: AssemblyCulture("")]
[assembly: Guid("018B6BE4-B525-4AC6-9568-BF8CC3660287")]
[assembly: ComVisible(false)]
```

To retrieve all attributes available to an assembly, call **GetCustomAttributes** method to obtain an enumerable list of **Attribute** objects. Following shows how to display the type of each available assembly attribute. Specific attributes can be fetched from an Assembly using **GetCustomAttribute** method and passing in attribute **Type**.

Displaying the types of all assembly attributes

```
IEnumerable<Attribute> attributes = a5.GetCustomAttributes();
foreach (Attribute attribute in attributes)
    Console.WriteLine("\t{0}", attribute.GetType());
```

Accessing an assembly attribute

```
var aa1 = (AssemblyTitleAttribute)a5.GetCustomAttribute(typeof(AssemblyTitleAttribute));
if (aa1 != null) Console.WriteLine(aa1.Title);
```

Using generic extension method

```
var aa2 = a5.GetCustomAttribute<AssembyVersionAttribute>();
if (aa2 != null) Console.WriteLine(aa2.Version);
```

2 Reflection

2.1 Base Classes & Interfaces

If you write code for a specific type, then that code would only work for that type and possibly derived types but would not work for other unrelated types. You would then have to write separate code for each type even though they have the same methods and properties.

Method that works for DebugLogger but not other loggers: Compatibility1

```
static void LogTo(DebugLogger logger) {
    logger.Message("Hello!");
    logger.Message("Goodbye!");
}
```

If you target the base class, then the same code can work with all types that derive directly or indirectly from the base class. However any class that only implements the **ILogger** interface but does not inherit from the base class will not be supported.

Method that works for all types derived from BaseLogger

```
static void LogTo(BaseLogger logger) {
    logger.Message("Hello!");
    logger.Message("Goodbye!");
}
```

If you target an interface, it then guarantees that all types implementing the interface will be compatible to the code regardless of the base class. This allows the same code to be applied to objects from many different types including future types.

Method that works with any type that implements ILogger

```
static void LogTo(ILogger logger) {
    logger.Message("Hello!");
    logger.Message("Goodbye!");
}
```

Compatibility can be checked at compilation time or runtime. When the exact type of class or interface is specified, the compiler will ensure that the object that you pass in is compatible to that type or interface. For example you can only pass objects to the following method where their types implement the **ICloneable** interface. Code trying to pass incompatible objects cannot be compiled.

Method that accepts only ICloneable objects

```
static object Copy(ICloneable obj) {
    return obj.Clone();     // calling Clone method through ICloneable interface
}
```

Passing items to method

```
var ob1 = Copy("Hello!");           // can compile because String type implements ICloneable
var ob2 = Copy(new ArrayList());    // can compile because ArrayList implements ICloneable
var ob3 = Copy(new int[4]));        // can compile because arrays implements ICloneable
var ob4 = Copy(DateTime.Now);       // cannot compile as DateTime does not support ICloneable
```

You can choose to do runtime compatibility check where you allow any kind of object to be passed in and use the **is** operator to verify compatibily. You can then type-cast the object to the base class or interface in order to use the object.

Runtime instead of compile-time checking

```
static object Copy(object obj) {
    if (obj is ICloneable) {
        ICloneable item = (Icloneable)obj;
        return item.Clone();
    }
    return null;    // null is default value if cannot clone
}
```

Passing items to method

```
var ob1 = Copy("Hello!");           // can compile and Clone() will be called
var ob2 = Copy(new ArrayList());    // can compile and Clone() will be called
var ob3 = Copy(new int[4]));        // can compile and Clone() will be called
var ob4 = Copy(DateTime.Now);       // can compile but result is null
```

If the purpose of using the **is** operator is so that you can use the object then it would be simpler to use the **as** operator instead for safe-casting. When you do safe-casting if the object is not compatible, there is no error and **null** will be returned. If not null then the object is compatible and you can use it.

Using safe-casting operator

```
static object Copy(object obj) {
    ICloneable item = obj as ICloneable;
    if (item != null) return item.Clone();
    return null;
}
```

Shorter version using ? operator

```
static object Copy(object obj) {
    ICloneable item = obj as ICloneable;
    return item != null ? item.Clone() : null;
}
```

2.2 Using Reflection

It is also possible that you can write code that can use any object without having any base class or interface by going through the reflection system. Reflection system work by using type information. You can retrieve and pass type information from a type by using **typeof** operator. You can also retrieve type information from any object using the **GetType** method. From type information, you can find out everything about the type and then use the type information to access objects.

To implement application framework libraries and smart client applications it is often required to write code that works with any type of object including types that are not even implemented yet. It is possible to retrieve meta-data information about types of objects during runtime and use reflection mechanism provided in .NET to interrogate type information and use the information to both instantiate and also access objects of that type. Add the following methods to **DebugHelper** in **Symbion** to facilitate the access of type-information; a feature called as reflection.

Methods to reflect members of a type: Symbion\DebugHelper.cs

```
public static void ShowFields(this Type type) {
    FieldInfo[] items = type.GetFields(); Console.WriteLine("\nFields:");
    foreach (FieldInfo item in items)
        Console.WriteLine("\t{0},{1}", item.Name, item.FieldType);
}
public static void ShowProperties(this Type type) {
    PropertyInfo[] items = type.GetProperties(); Console.WriteLine("\nProperties:");
    foreach (PropertyInfo item in items)
        Console.WriteLine("\t{0},{1}", item.Name, item.PropertyType);
}
public static void ShowEvents(this Type type) {
    EventInfo[] items = type.GetEvents(); Console.WriteLine("\nEvents:");
    foreach (EventInfo item in items)
        Console.WriteLine("\t{0},{1}", item.Name, item.EventHandlerType);
}
public static void ShowConstructors(this Type type) {
    ConstructorInfo[] items = type.GetConstructors(); Console.WriteLine("\nConstructors:");
    foreach (ConstructorInfo item in items) {
        Console.WriteLine("\t{0}", item.Name);
        ParameterInfo[] parameters = item.GetParameters();
        foreach (ParameterInfo param in parameters)
            Console.WriteLine("\t\t{0},{1}", param.Name, param.ParameterType);
    }
}
public static void ShowMethods(this Type type) {
    MethodInfo[] items = type.GetMethods(); Console.WriteLine("\nMethods:");
    foreach (MethodInfo item in items) {
        Console.WriteLine("\t{0},{1},{2}",
        item.Name, item.ReturnType, item.IsStatic ? "(S)" : "");
        ParameterInfo[] parameters = item.GetParameters();
        foreach (ParameterInfo param in parameters)
            Console.WriteLine("\t\t{0},{1}", param.Name, param.ParameterType);
    }
}
```

Method to reflect a type

```
public static void ShowTypeInfo(this Type type) {
    Console.WriteLine("Name: {0}", type.Name);
    Console.WriteLine("Namespace: {0}", type.Namespace);
    Console.WriteLine("FullName: {0}", type.FullName);
    Console.WriteLine("BaseType: {0}", type.BaseType);
    Console.WriteLine("Assembly: {0}", type.Assembly.FullName);
    Console.WriteLine("IsEnum: {0}", type.IsEnum);
    Console.WriteLine("IsClass: {0}", type.IsClass);
    Console.WriteLine("IsValueType: {0}", type.IsValueType);
    Console.WriteLine("IsInterface: {0}", type.IsInterface);
    if (type.IsClass || type.IsValueType) {
        type.ShowFields();
        type.ShowProperties();
        type.ShowEvents();
        type.ShowConstructors();
        type.ShowMethods();
    }
}
```

Also note that custom attributes can be accessed for each member when you call the **GetCustomAttributes** and **GetCustomAttribute** methods on any member including also the **Type** and **Assembly**.

You can get **Type** information by using **typeof** operator on a specific type or use the **GetType** method on value or object. Reflection is not just to access type-information. You can also write code that can dynamically create and use objects of any type. This is important for frameworks that are built to be-reusable across multiple applications where every application will have its own custom types, so the framework has to be able to adapt and work with many types. The following section shows how to use an object through reflection. You can obtain type-information from an object / value by calling **GetType**. You can use reflection to retrieve specific type members by name. The reflection can then be used to dynamically access the class or object.

Obtaining type information at runtime: Reflection1

```
Type t1 = "This is a string".GetType();  // getting type information from type
Type t2 = typeof(DebugHelper);           // getting type information from type
t1.ShowTypeInfo();
t2.ShowTypeInfo();

// Getting type information from an object or interface
var item = new LoanAccount(100, "ABC Trading", 50000m, 0.06m, 5);
object o1 = item;
var t1 = item.GetType();
```

Accessing a property

```
PropertyInfo p1 = t1.GetProperty("Balance");
if (p1 == null) { Console.WriteLine("No Balance property!"); return; }
if (p1.PropertyType != typeof(decimal)) { Console.WriteLine("Wrong property type!"); return; }
decimal balance = (decimal)p1.GetValue(o1, null);
balance -= 10000m; p1.SetValue(o1, balance, null);
Console.WriteLine("{0},{1}", balance, item.Balance);
```

Invoking a method

```
Type[] parameterTypes = { typeof(string) };
MethodInfo m1 = t1.GetMethod("ToString", parameterTypes);
if (m1 == null) { Console.WriteLine("No ToString(string) method!"); return; }
if (m1.ReturnType != typeof(string)) { Console.WriteLine("Not string method!"); return; }
object[] parameters = { "ID={0}\nName={1}\nBalance={2}" };
string result = (string)m1.Invoke(o1, parameters);
Console.WriteLine(result);
```

To simplify operations you can also call **InvokeMember** directly on a type to perform the same operations. The following shows how to do this as well as invoking static methods and calling the constructor.

Accessing properties and methods using InvokeMember

```
balance = (decimal)t1.InvokeMember(
    "Balance", BindingFlags.GetProperty, null, o1, null);
Console.WriteLine(balance);
result = (string)t1.InvokeMember(
    "ToString", BindingFlags.InvokeMethod, null, o1, parameters);
Console.WriteLine(result);
```

Calling static methods

```
parameters = new object[] { 60000m, 0.07m, 5 };
decimal payment = (decimal)t1.InvokeMember("GetPayment",
    BindingFlags.InvokeMethod, null, null, parameters);
Console.WriteLine(payment);
```

Creating new instances and calling constructors

```
parameters = new object[] {200, "XYZ Trading", 60000m, 0.07m, 5 };
object o2 = t1.InvokeMember(null, BindingFlags.CreateInstance,
    null, null, parameters);
Console.WriteLine(o2 is LoanAccount);
```

However it would be far simpler to use **Activator** to instantiate objects from types as shown below. You can easily call the default and custom constructors on any type.

Using Activator to create instances

```
object o3 = Activator.CreateInstance(t1);
object o4 = Activator.CreateInstance(t1, parameters);
object o5 = Activator.CreateInstance(t1, 210, "ZZZ Trading", 52000m, 0.05m, 6);
Console.WriteLine(o3 is LoanAccount);
Console.WriteLine(o4 is LoanAccount);
Console.WriteLine(o5 is LoanAccount);
Console.WriteLine(o4);
Console.WriteLine(o5);
```

Reflection code can be tedious and complex to write. If there is a easier way for you to write code that can access many different types of objects like using base classes or interfaces, then those will be simpler solutions. However sometimes reflection may be the only way to resolve certain object access issues like anonymous objects.

2.3 Anonymous & Dynamic Objects

In C# 3.0 you are able to create instances of anonymous types by just using the **new** keyword without the type name. You can provide a list of property initializers to store information into the anonymous object. The type of object is known in the method that instantiated the object but will not be known outside of the method. In order for other methods to access the content of the object, you will have to use reflection as shown below.

Display properties of any type: Symbion\DebugHelper.cs

```
public static void DisplayProperties(this object obj) {
    PropertyInfo[] props = obj.GetType().GetProperties();
    foreach (PropertyInfo prop in props) Console.WriteLine("{0}={1}",
            prop.Name, prop.GetValue(obj, null));
}
```

Instancing an anonymous type: Anonymous1

```
static void showAccount(object obj) {
    obj.DisplayProperties();
}

static void main() {
    var account = new { ID = 101, Name = "ABC Trading", Balance = 50000m };
    Console.WriteLine(account.ID);
    Console.WriteLine(account.Name);
    Console.WriteLine(account.Balance);
    showAccount(account);
}
```

In C# 4.0 you can now use dynamic objects. Dynamic objects are much slower to use and there is no intellisense whatsoever. You can assume that the object has a certain feature and use the feature directly in the code without type information. If the object at runtime does not have that feature, only a runtime error will occur which you can catch and handle. The compiler will generate different code when accessing a dynamic object and does not use reflection.

Using dynamic objects

```
static void showAccount(dynamic obj) {
    Console.WriteLine(obj.ID);
    Console.WriteLine(obj.Name);
    Console.WriteLine(obj.Balance);
}
```

Certain methods may be quite complex to implement based on type compatibility and restrictions. However they are very simple to implement when using dynamic objects. The following method can be used to attempt to add anything to anything and return anything. Whether it works or not depends not on the compiler but the actual objects or values that you pass in at runtime.

A dynamic method to add anything to anything and return anything

```
static dynamic Add(dynamic v1, dynamic v2) {
    return v1 + v2;
}
```

Passing different values and objects to a dynamic method

```
Console.WriteLine(Add(99, 66));
Console.WriteLine(Add(1.99, 66.1));
Console.WriteLine(Add('C', 2));
Console.WriteLine(Add("Hello!", "Goodbye!"));
Console.WriteLine(Add(DateTime.Now, new TimeSpan(1, 2, 3, 4)));
```

3 Task Parallel Library

3.1 Data Parallelism

Multi-threading does not really improve performance if you only have one processor. It just allows you to fully utilize the processor. When a thread is waiting for something it is not using CPU time, than that time can be utilized by another thread. However, if you do have multiple processors performance can be improved by distributing threads across all the processors. One thread can only run on one processor so you definitely need to create multiple threads to fully utilize all available processors. However it has always been very difficult to program a single operation to use multiple threads but in .NET 4.0, Microsoft has implement the Task Parallel Library (TPL) to simplify this. The following is a method that runs completely on one thread.

Operation that runs on one thread: Parallel1\Program.cs

```
static List<string> list1 = new List<string>();
static int ScanDirectory1(string path) {
    try { var files = Directory.GetFiles(path);
        foreach (string file in files) list1.Add(file);
        var directories = Directory.GetDirectories(path);
        int count = files.Length;
        foreach (var directory in directories)
            count += ScanDirectory1(directory);
        return count;
    } catch (Exception) { return 0; }
}
```

The above method scans a directory and all sub-directories on one thread to collect a list of files and return the number of files collected. It would be faster if we can use a separate thread to scan through each sub-directory and these threads is distributed across multiple processors. This can be easily done using TPL by replacing a **foreach** with **Parallel.ForEach** method instead as shown below.

Operation that runs across multiple threads: Parallel1\Program.cs

```
static List<string> list2 = new List<string>();
static int ScanDirectory2(string path) {
    try {   var files = Directory.GetFiles(path);
        foreach (string file in files) list2.Add(file);
        var directories = Directory.GetDirectories(path);
        int count = files.Length;
        Parallel.ForEach(directories, directory =>
            count += ScanDirectory2(directory));
        return count;
    }   catch (Exception) { return 0; }
}
```

We will now measure the time taken to run both methods using **Stopwatch**. The time is consistent for the first method regardless of how many processors you have while the second method will scale automatically to the number of available processors.

Timing synchronous and asynchronous operations

```
static void Main() {
    var watch1 = Stopwatch.StartNew();
    int count1 = ScanDirectory1("C:\\Windows");
    watch1.Stop();
    var watch2 = Stopwatch.StartNew();
    int count2 = ScanDirectory2("C:\\Windows");
    watch2.Stop();
    Console.WriteLine(watch1.ElapsedMilliseconds);
    Console.WriteLine(watch2.ElapsedMilliseconds);
    Console.WriteLine(count1);
    Console.WriteLine(count2);
    Console.WriteLine(list1.Count);
    Console.WriteLine(list2.Count);
}
```

3.2 Concurrency

While the parallel operation is definitely faster, the results are actually incorrect. This is due to concurrency problems that occur when we try to update the same collection or field from multiple threads. All of the collection types in **System.Collections** and **System.Collections.Generic** are not thread-safe. Adding items into the collection by multiple threads at the same time can corrupt the internal structure of the collection. Results can range from inaccurary of items added or even crashing the program. You can synchronize non thread-safe operations by using **lock**.

Synchronize updating of a shared collection

```
foreach (string file in files) lock(list2) list2.Add(file);
```

Alternatively you use a thread-safe collection from **System.Collections.Concurrent**. Methods in these collections will perform internal synchronization. They will be slower but the results will be accurate.

Replace List<T> with ConcurrentBag<T>

```
static ConcurrentBag<string> list2 = new ConcurrentBag<string>();
```

While the collection is now accurate, the file **count** is not. You can use **lock** keyword to synchronize updating of the counter. However since updating counters is common in multi-threading operations, you can can use a special **Interlocked** class to update counters instead.

```
Parallel.ForEach(directories, directory =>
    Interlocked.Add(ref count, ScanDirectory2(directory)));
```

3.3 Code Parallelism

While **Parallel.ForEach** is a asynchronous version of **foreach**, you can also run other statements asychronously by using **Parallel.Invoke**. It can accept a parameter array of **Action** delegates to run code. Multiple threads will be used to run the delegates on multiple processors.

Performing operations synchronously: Parallel2\Program.cs

```
static void Main() {
    Console.WriteLine("Task 1"); Thread.Sleep(4000);
    Console.WriteLine("Task 2"); Thread.Sleep(4000);
    Console.WriteLine("Task 3"); Thread.Sleep(4000);
    Console.WriteLine("Finish");
}
```

Performing operations asynchronously

```
Parallel.Invoke(
    () => { Console.WriteLine("Task 1"); Thread.Sleep(4000); },
    () => { Console.WriteLine("Task 2"); Thread.Sleep(4000); },
    () => { Console.WriteLine("Task 3"); Thread.Sleep(4000); }
);
Console.WriteLine("Finish");
```

Since each parameter of the parameter array is just an **Action** delegate, any complex tasks can implemented as separate methods and pass as individual parameters or as an array.

Task code implemented in separate methods

```
static void Task1() { Console.WriteLine("Task 1"); Thread.Sleep(4000); }
static void Task2() { Console.WriteLine("Task 2"); Thread.Sleep(4000); }
static void Task3() { Console.WriteLine("Task 3"); Thread.Sleep(4000); }
```

Pass as individual parameters

```
Parallel.Invoke(Task1, Task2, Task3);
```

Pass in an array

```
Action[] tasks = { Task1, Task2, Task3 };
Parallel.Invoke(tasks);
```

3.4 Creating Tasks

Parallel class create **Task** objects to allocate threads to run code. You can also use this class directly. You can call **Start** to begin running each task and call **Wait** to wait for the task to be completed. There is also a generic version of **Task** that allows execution of tasks that has a result.

Creating and using Task objects: Tasks1

```
Task task1 = new Task(() => { Console.WriteLine("Task 1"); Thread.Sleep(4000); });
Task task2 = new Task(() => { Console.WriteLine("Task 2"); Thread.Sleep(4000); });
Task task3 = new Task(() => { Console.WriteLine("Task 3"); Thread.Sleep(4000); });
task1.Start(); task2.Start(); task3.Start();
task1.Wait(); task2.Wait(); task3.Wait();
Console.WriteLine("Finish");
```

Additional Task features:

```
Task.WaitAll(task1,task2,task3);                 // wait for multiple tasks to complete
Task.WaitAny(task1,task2,task3);                 // wait for one of the tasks to complete
Task tasks1 = Task.WhenAll(task1,task2,task3);   // combining WaitAll tasks into one
Task tasks2 = Task.WhenAny(task1,task2,task3);   // combining WaitAny tasks into one
tasks1.Wait();                                   // wait for multiple tasks to complete
tasks2.Wait();                                   // wait for one of the tasks to complete
```

Returning results from a Task

```
Task<int> task4 = new Task<int>(() => {
    Console.WriteLine("Task 4"); Thread.Sleep(4000); return 123; });
task4.Start(); task4.Wait();
Console.WriteLine(task4.Result);
```

3.5 Task Cancellation

Aborting a thread can be dangerous as you have no idea what the thread was doing at that time. It could hang the system or leave it in an unstable state. It will be better if the task can choose the best location where the thread is not performing any critical operation and thus is safe to be aborted. TPL provides a developer controlled method to cancel tasks. Create a **CancellationTokenSource** where the **CancellationToken** can be obtained. Call **Cancel** method on the source to request for cancellation.

Obtaining a CancellationToken: Tasks2

```
var cs = new CancellationTokenSource();
var ct = cs.Token;
```

Using CancellationSource to cancel:

```
var task1 = new Task(() => {
    while (true) {
        var input = Console.ReadKey(true);
        if (input.Key == ConsoleKey.Escape) {
            cs.Cancel();
            break;
        }}});
```

You can associate the CancellationToken with any Task you create. You can check the **IsCancellationRequested** property at a safe place in the code to determine if there is a cancel request or throw an exception with the **ThrowIfCancellationRequested** method to stop the current task if cancellation has been requested. To determine if an exception that stopped a task was due to a cancellation check the task's **IsCanceled** property when an exception has been caught.

Stop task if cancellation requested:

```
var task2 = new Task(() => {
    while (true) {
        Thread.Sleep(2000);
        Console.WriteLine("Task still running...");
        if (ct.IsCancellationRequested) {
            Console.WriteLine("Task cancelled.");
            break;
        }
    }}, ct);// associated cancellation token with Task

task2.Start();
task1.Start();
task2.Wait();
```

Generating exception on cancel:

```
while (true) {
    Thread.Sleep(2000);
    Console.WriteLine("Task still running...");
    ct.ThrowIfCancellationRequested();
}
```

Determining whether task has been cancelled

```
try {
    task2.Start();
    task1.Start();
    task2.Wait();
    Console.WriteLine("Task completed.");
}
catch (Exception) {
    if (task2.IsCanceled) Console.Write("Task was cancelled.");
    else Console.WriteLine("Error occurred in task.");
}
```

3.6 Async & Await

To run code asynchronous without blocking the UI in a windows application you need to first encapsulate the task into an asynchronous method.You can then call it from UI thread with or without using **await** keyword. The method that uses **await** must be be marked **async** as it is also required to run asynchronously so that the primary thread or the UI thread is not blocked.

Implementing UI non-blocking asynchronous operations

```
void Task<string> RunTaskAsync() {   // implementing an asynchronous method
    var task = new Task(() => { Thread.Sleep(6000); return "Done!"; });
    task.Start(); return task;
}

private async void btnRunTask_Click(sender obj, EventArgs e) {
    btnRunTask.IsEnabled = false;
    txtResult.Text = await RunTaskAsync();
    btnRunTask.IsEnabled = true;
}
```