|  |
| --- |
|  |
| C# & the CLR |
| The Runtime |
|  |
|  |
|  |

|  |
| --- |
|  |

# 

# C# and the CLR

## Objectives

In this lab we’ll build a small console mode program and use reflection to find all the public types in all the assemblies referenced by our project. We’ll create an instance of each type (the types we are allowed to create) and see the difference between serial and parallel execution of the code.

## Part I – Serial Execution

1. In Visual Studio, create a new Console Application named **CSCLR** in the before directory of this lab.
2. In Program.cs, make sure you import the following namespaces.

using System;  
using System.Collections.Concurrent;  
using System.Diagnostics;  
using System.Reflection;

using System.Threading.Tasks;

1. As the first line of code inside the Main method, create a new instance of a ConcurrentStack to hold objects.

ConcurrentStack<object> stack = new ConcurrentStack<object>();

ConcurrentStack is a class that implements a thread-safe stack, meaning you can add and remove items from the stack using multiple threads, and the stack itself will maintain locks to prevent data corruption. The <object> part of the statement is a generic type parameter that tells the compiler we want the stack to hold any type of object. We’ll be taking a closer look at generics in the next module.

1. Use the System.Reflection.Assembly class to load the names of all referenced assemblies (you’ll need two method calls to do this).

AssemblyName[] names = Assembly.GetExecutingAssembly()  
                               .GetReferencedAssemblies();

1. Use the static StartNew method of the System.Diagnostics.Stopwatch class to create a new timer.

Stopwatch watch = Stopwatch.StartNew();

1. Write a for loop that will iterate over the assembly names collected earlier.

for(int i = 0; i < names.Length; i++)  
{  
      
}

1. Inside the loop, use Assembly.Load to turn each assembly name into an assembly reference.

Assembly assembly = Assembly.Load(names[i]);

1. Use the reflection API to obtain an array of all types in the assembly.

Assembly assembly = Assembly.Load(names[i]);  
**Type[] types = assembly.GetTypes();**

1. Write another loop to iterate over the type array.

for(int i = 0; i < names.Length; i++)  
{  
    Assembly assembly = Assembly.Load(names[i]);  
    Type[] types = assembly.GetTypes();  
**for(int j = 0; j < types.Length; j++)  
    {  
          
    }**}

Notice this is a nested loop.

1. Inside the second loop, write an if statement to check if the type is public.

for(int j = 0; j < types.Length; j++)  
{  
 **if(types[j].IsPublic)  
    {  
          
    }**}

At this point we want to construct an instance of the type and add it to our stack. However, we don’t know if the type even has a public constructor. We’ll use some more reflection to ensure we can create the type.

1. Use the reflection API to retrieve a list of constructors for the type.

if(types[j].IsPublic)  
{  
**ConstructorInfo[] constructors = types[j].GetConstructors();**}

1. Write a foreach loop to iterate over the constructor information.

ConstructorInfo[] constructors = types[j].GetConstructors();  
**foreach(ConstructorInfo constructor in constructors)  
{  
      
}**

1. Inside the foreach loop, use the code shown below to make sure the type is a type we can instantiate.

foreach(ConstructorInfo constructor in constructors)  
{  
**if (constructor.GetParameters().Length == 0 &&  
         types[j].GetGenericArguments().Length == 0)  
        {  
          
        }**  
}

The first test in the if statement makes sure we have a default constructor (a constructor that takes 0 parameters). The second test ensures the type doesn’t require a generic argument to instantiate (we’ll look at generics in the next lab).

1. Inside the if check, use the Activator class to construct an instance of the type. Add the newly constructed object to the stack.

if (constructor.GetParameters().Length == 0 &&  
     types[j].GetGenericArguments().Length == 0)  
    {  
        stack.Push(Activator.CreateInstance(types[j]));  
    }

1. Outside of all the looping, use the watch reference to determine how many milliseconds have elapsed. Store this value into a variable.

long elapsedTime = watch.ElapsedMilliseconds;

1. Using a foreach statement and Console.WriteLine, write out the type name of each object in the stack.

foreach (var type in stack)  
{  
    Console.WriteLine(type.GetType().Name);  
}

1. Also write out the total number of objects in the stack, and the elapsed time.

Console.WriteLine("Total objects: {0}", stack.Count);  
Console.WriteLine("Elapsed time: {0}", elapsedTime);

1. Run the application and record the elapsed time. You might want to run the application 4-5 times to make sure the number if stables.

## Part II – Parallel Execution

1. Replace the outermost for loop with a call to Parallel.For(). The code will look like the following.

Parallel.For(0, names.Length, (i) =>  
{  
    Assembly assembly = Assembly.Load(names[i]);  
    Type[] types = assembly.GetTypes();  
    for (int j = 0; j < types.Length; j++)  
    {  
        if (types[j].IsPublic)  
        {  
            ConstructorInfo[] constructors = types[j].GetConstructors();  
            foreach (ConstructorInfo constructor in constructors)  
            {  
                if (constructor.GetParameters().Length == 0 &&  
                     types[j].GetGenericArguments().Length == 0)  
                {  
                    stack.Push(Activator.CreateInstance(types[j]));  
                }  
            }  
        }  
    }  
});

Note 1: Notice the ); at the end of the code listing.

Note 2: The (i) => syntax is a C# *lambda expression*. If you haven’t worked with lambda expressions you’ll want to look at the C# and LINQ module of the LINQ Fundamentals course.

1. Run the code again and measure the elapsed time. Did the time improve?
2. Now replace the inner for loop with a call to Parallel.For. The code will look like the following.

Parallel.For(0, names.Length, (i) =>  
{  
    Assembly assembly = Assembly.Load(names[i]);  
    Type[] types = assembly.GetTypes();  
    Parallel.For(0, types.Length, (j) =>  
    {  
        if (types[j].IsPublic)  
        {  
            ConstructorInfo[] constructors = types[j].GetConstructors();  
            foreach (ConstructorInfo constructor in constructors)  
            {  
                if (constructor.GetParameters().Length == 0 &&  
                     types[j].GetGenericArguments().Length == 0)  
                {  
                    stack.Push(Activator.CreateInstance(types[j]));  
                }  
            }  
        }  
    });  
});

1. Run the application and measure the elapsed time. Did the time improve?

You might notice the time gets worse. In some cases, adding additional parallelization and threads creates more overhead than it is worth. You need to take measurements to ensure you are improving the code. Nevertheless, we’ll try this once more.

1. Replace the inner foreach loop with a call to Parallel.ForEach.

Parallel.For(0, names.Length, (i) =>  
{  
    Assembly assembly = Assembly.Load(names[i]);  
    Type[] types = assembly.GetTypes();  
    Parallel.For(0, types.Length, (j) =>  
    {  
        if (types[j].IsPublic)  
        {  
            ConstructorInfo[] constructors = types[j].GetConstructors();  
            Parallel.ForEach(constructors, (constructor) =>  
            {  
                if (constructor.GetParameters().Length == 0 &&  
                     types[j].GetGenericArguments().Length == 0)  
                {  
                    stack.Push(Activator.CreateInstance(types[j]));  
                }  
            });  
        }  
    });  
});

1. Run the application again and measure the results.

# Conclusion

Congratulations! You’ve combined threading and reflection to get a feel for some of the core services available from the CLR.