|  |
| --- |
|  |
| C# 4: Dynamic C# |
| Building Objects with Dynamic Behavior |
|  |
|  |
|  |

# 

# C#: Dynamic C#

## Objectives

After completing this lab, you should understand how to:

* Use the dynamic keyword to bypass static type checking
* Build a object with dynamic behavior by deriving from DynamicObject

## DynamicXml

1. Open the solution file in the before folder.
2. Open the BuildXml.cs file

The project you have opened is a unit testing project. There are tests inside this BuildXml C# file, but they are all commented out. The goal is to uncomment the tests one at a time and make each one pass. If you wish, you can disregard this lab manual and let the unit tests guide your work.

1. Uncomment the Step1 test. Implement everything you need to make the test pass. You’ll need a static method on the DynamicXml class named Parse – and it’s return type should be **dynamic**.

This first test will require you to create a DynamicXml class definition. The class will need a static Parse method – can you figure out the parameters and return type of the Parse method? You’ll also need to derive from a very special base class. The bare minimum required to make this test pass is shown below.

public class DynamicXml : DynamicObject

{

public static dynamic Parse(string xml)

{

return new DynamicXml();

}

}

1. Uncomment the Step2 test. Implement everything you need to make the test pass. You’ll need to override the proper method on the DynamicXml class and respond to the get request for “People”.

For the first test we used the bare minimum code to make the test pass, so this second test is a little more challenging. We’ll need to parse and save the incoming xml before we can respond to the request for “People”. One approach is to use the following code.

public class DynamicXml : DynamicObject

{

private readonly dynamic \_xml;

public static dynamic Parse(string xml)

{

return new DynamicXml(XDocument.Parse(xml));

}

public DynamicXml(dynamic xml)

{

\_xml = xml;

}

public override bool TryGetMember(GetMemberBinder binder,

out object result)

{

result = \_xml.Element(binder.Name);

return result != null;

}

}

Notice how every result we return must be wrapped inside of DynamicXml to maintain the dynamic behavior.

1. Now move on the step 3 test. If you run the test you’ll see an exception during the test execution. You’ll need the DynamicXml class to implement the IEnumerable interface. One possible implementation is shown below.

public IEnumerator GetEnumerator()

{

foreach(var childElement in \_xml.Elements())

{

yield return new DynamicXml(childElement);

}

}

1. Finally, uncomment the step 4 test. In order to get the test to pass you’ll need to make sure TryGetMember can work with elements or attributes, perhaps by using some if/else logic. A full implementation of the DynamicXml class which passes all tests is shown below.

public class DynamicXml : DynamicObject, IEnumerable

{

public static dynamic Parse(string xml)

{

return new DynamicXml(XDocument.Parse(xml));

}

public DynamicXml(dynamic xElement)

{

this.xml = xElement;

}

public override bool TryGetMember(GetMemberBinder binder,

out object result)

{

if(xml.Element(binder.Name) != null)

{

result = new DynamicXml(xml.Element(binder.Name));

return true;

}

else if(xml.Attribute(binder.Name) != null)

{

result = xml.Attribute(binder.Name).Value;

return true;

}

result = null;

return false;

}

public IEnumerator GetEnumerator()

{

foreach (var element in xml.Elements())

{

yield return new DynamicXml(element);

}

}

private dynamic xml;

}

# Conclusion

Congratulations! In this lab we used the new dynamic features of C# to build a flexible XML parser. The combination of the dynamic type with the DynamicObject class enabled us to write code that is easier on the eyes than traditional XML processing code.