CLI dotnet

.NET CLI is a C# port of the Commons CLI API. The project is hosted on SourceForge and maintained by Schley Andrew Kutz.

For more information see http://sf.net/projects/dotnetcli

News

.NET CLI and Object.MemberwiseClone

A helpful reader pointed out that .NET **does** have a shallow clone method defined on the Object class called **MemberwiseClone**. I have updated the Option class to use this method (that I somehow stupidly forgot about) and removed the requirement that anything deriving from Option must implement ICloneable or be serializable.

C# port now referred to as .NET CLI

The port is called .NET CLI, and the unix name on SourceForge is dotnetcli. The project has a default namespace of net.sf.dotnetcli.

C# port now hosted on SourceForge

The URL is http://sf.net/projects/dotnetcli/.

Updates on C# port - Unit testing almost complete, and changes

The unit tests from 1.1 of CLI have almost all been ported to the C# version. I am happy to say that the port passes 45/45 of the tests ported so far. The following additional changes have been implemented in the C# port:

- In C# you cannot access a static method, field, or property accessor from an instance reference (static or otherwise), hence the following code is illegal:

```
Option timeLimit = OptionBuilder
        .withLongOpt("limit")
        .hasArg()
        .withValueSeparator()
        .withDescription("Set time limit for execution, in mintues")
        .create("l");
```

So, in order to maintain elegance I have come up with a solution. I have changed OptionBuilder's static methods to instance methods, but created a static property accessor called 'Factory'. Factory is defined as such:

```
/// <summary>
///                Returns a static instance of OptionBuilder.
/// </summary>
public static OptionBuilder Factory
{
        get { return instance; }
}
```

So the original code now works with one small variation:

```
Option timeLimit = OptionBuilder.Factory
        .withLongOpt("limit")
        .hasArg()
        .withValueSeparator()
        .withDescription("Set time limit for execution, in mintues")
        .create("l");
```

- In C# all classes derive from Object, just like Java. Even primitive types are automatically boxed and unboxed into Object-derived subclasses, just like Java. For a class to be cloneable it must implement the interface ICloneable and the method 'public object Clone()', just like Java. However, unlike Java, Object does not implement a shallow clone method. This means that the logic you implement in your own clone method cannot call the super (or base as it is in C#) class's Clone method to get a typed, shallow clone. This would not necessarily be a big deal as you could just create a new class that you are attempting to clone, but if you do this you would have to be able to clone its properties (assuming they are publicly accessible or can be set via a constructor) manually – a cumbersome task.

check this.MemberwiseClone()

You see the problem. Option is cloneable, and it relies on the Object's clone method. I toyed around porting this functionality via two methods. The first method looked like this:

```
object clone = Activator.CreateInstance( this.GetType(), null );
( clone as Option ).m_alist_values = new List<string>( m_alist_values );
return clone;
```

The above code will not work in all cases, hence I discarded it. The problem is that the Activator.CreateInstance method's second parameter requires either a null value for a parameterless constructor, or a parameter array to pass to the constructor of the type you are attempting to create. As seen in OptionTest.java (and now OptionTest.cs), the DefaultOption class that extends Option does not implement a parameterless constructor, and in fact does not even override all of Option's constructors, just one. So, while using the CreateInstance method *would* result in the proper object type once cloned, it is cumbersome (although possible) to know the right constructor parameters to pass to the CreateInstance method. Even if you do figure out the parameters needed, you still have the task of cloning the new Option's properties, some of which are not publicly scoped.

Instead, I am serializing the object to be cloned and deserializing it into a new copy. Like so:

```
BinaryFormatter bf = new BinaryFormatter();
MemoryStream memStream = new MemoryStream();
bf.Serialize( memStream, this );
memStream.Flush();
memStream.Position = 0;
object clone = ( bf.Deserialize( memStream ) );
( clone as Option ).m_alist_values = new List<string>( m_alist_values );
return clone;
```

This method is certainly not as efficient as a simple object creation, but it guarantees future compatibility with new properties that Option may receive. The only caveat is that the Option class, and any class that extends it, must implement the attribute [Serializable] so that it can be serialized:

```
[Serializable]
public class MyOption : Option
{
...
}
```

Release 1.1 Ported to C#

Version 1.1 of the CLI library has been ported to C# (Homepage) and is available at http://code.lostcreations.com/browser/trunk/lib/csharp/Cli/. You can download the sources from the subversion repository with the following command:

```
svn co http://svn.lostcreations.com/code/trunk/lib/csharp/Cli/
```

The following changes were made to CLI while porting:

- Java's iterator is bi-directional while .NET's enumerator is simple, and uni-directional. I created a class, BidirectionalEnumerator<T> to match the functionality present in Java.
- Getters and setters have become property accessors. For example, instead of getWidth() and setWidth(), there is now Width { get; set; }.
- Method names now begin with upper case.
- Class level variables are now prefixed with "m_".
- Interfaces now begin with upper case "I". Ex. CommandLineParser is now ICommandLineParser.
- I've used generics where possible instead of Objects. Ex. An ArrayList of Object types intended to hold strings is now List<string>.
- I've used Dictionary<Tk,Tv> for a hash table.
- In Java, the method substring is defined as func(int begin, int end) where end is exclusive and the length to splice is a result of end - begin. In .NET Substring is func(int begin, int length) where length is inclusive. Instead of changing your math I instead created a static class called JavaPorts and have created the method JavaPorts.Substring(string value, int beginIndex, int endIndex) that functions as Java's substring.

There has only been minimal bug testing performed on the port, as it was ported to serve a particular need. Please submit bug reports as you find them, or view a list of outstanding issues.