Kicking around a function that formats stuff

devblogs.microsoft.com/oldnewthing/20141110-00

November 10, 2014

Raymond Chen

Today's "Little Program" is really a "Little Puzzle" that got out of hand.

This started out as a practical question: This code fragment screams out for some sort of simplification. (I've changed the names of the classes.)

```
class FrogProperty
{
 public string Name { get; private set; }
 public string Value { get; private set; }
 ...
}
class ToadProperty
{
 public string Name { get; private set; }
 public string Value { get; private set; }
 ...
}
var frogStuff = new List<string>();
foreach (var frogProp in FrogProperties) {
  frogStuff.Add(string.Format("{0}: {1}", frogProp.Name, frogProp.Value));
}
frogStuff.Sort();
Munge(frogStuff);
var toadStuff = new List<string>();
foreach (var toadProp in ToadProperties) {
  toadStuff.Add(string.Format("{0} = {1}", toadProp.Name, toadProp.Value));
}
toadStuff.Sort();
Munge(toadStuff);
var catStuff = new List<string>();
foreach (var cat in Cats) {
  catStuff.Add(string.Format("{0}", cat.Name));
}
catStuff.Sort();
Munge(catStuff);
var dogStuff = new List<string>();
foreach (var dogProp in DogProperties) {
  dogStuff.Add(string.Format("{0} {1}", dogProp.Name, dogProp.Value));
}
dogStuff.Sort();
Munge(dogStuff);
...
```

Clearly, the pattern is

```
var stuff = new List<string>();
foreach (var thing in thingCollection) {
 stuff.Add(string.Format(formatstring, thing.Name, [optional: thing.Value]));
}
stuff.Sort();
Munge(stuff);
```

Everything here is pretty straightforward, except for the `string.Format` part. Can we write a function that takes a `thing` and formats it in a somewhat flexible manner?

Let's start with the `Name`-and-`Value` cases. We might try something like this:

```
public static string FormatNameValue<T>(this T t, string format)
{
 return string.Format(format, t.Name, t.Value);
}
```

But then we'd run into trouble, because there is no constraint on `T`, so the compiler will complain, "I don't know how to get a `Name` or a `Value` from an `object`."

And since `FrogProperty` and `ToadProperty` do not have a common base class, you're kind of stuck.

One way out would be to use the new `dynamic` type:

```
public static string FormatNameValue<T>(this T t, string format)
{
 dynamic d = t;
 return string.Format(format, d.Name, d.Value);
}
```

But that won't work in the `Name`-only case:

```
cat.FormatNameValue("{0}");
```

The `cat` object has a `Name` but no `Value`. The attempt to read the `Value` will raise an exception (even though it is never consumed by the format).

Maybe we can turn to reflection.

```
public static string FormatNameValue<T>(this T t, string format)
{
 return string.Format(format,
                      typeof(T).GetProperty("Name").GetValue(t, null),
                      typeof(T).GetProperty("Value").GetValue(t, null));
}
```

This still raises an exception if there is no `Value`, but we can detect the missing `Value` before we run into trouble with it.

```
static object GetPropertyOrNull<T>(this T t, string prop)
{
 var propInfo = typeof(T).GetProperty(prop);
 return propInfo == null ? null : propInfo.GetValue(t, null);
}
public static string FormatNameValue<T>(this T t, string format)
{
 return string.Format(format,
                      t.GetPropertyOrNull("Name"),
                      t.GetPropertyOrNull("Value"));
}
```

Okay, now we're getting somewhere.

But before getting to deep into this exercise, I should point out that another way to solve this problem is to turn it inside-out. Instead of making the munger understand all of the different objects, why not make each object understand munging?

```
class FrogProperty : IFormattable
{
 public string Name { get; private set; }
 public string Value { get; private set; }
 public override ToString(string format, IFormatProvider formatProvider)
 {
  switch (format) {
  case "Munge":
   return string.Format(formatProvider,"{0}: {1}", Name, Value);
  default:
   return ToString(); // use object.ToString();
  }
 }
}
class Cat : IFormattable
{
 public string Name { get; private set; }
 public override ToString(string format, IFormatProvider formatProvider)
 {
  switch (format) {
  case "Munge":
   return string.Format(formatProvider,"{0}", Name);
  default:
   return ToString(); // use object.ToString();
  }
 }
}
```

The generic helper function would then be

```
var stuff = new List<string>();
foreach (var thing in thingCollection) {
 stuff.Add(string.Format("{0:Munge}", thing);
}
stuff.Sort();
Munge(stuff);
```

Okay, fine, rain on my little puzzle parade.

Let's ignore this very useful advice and proceed ahead with our puzzle, because *we're determined to see how far we can go, even if it's in the wrong direction*.

Now that we have `FormatNameValue`, we might say, "What about generalizing to cases where we want properties other than `Name` and `Value`?" One design would be to pass in a format string and list of properties you want to fill in:

```
thing.FormatProperties("{0}: {1} (modified by {2})",
                       "Name", "Value", "ModifiedBy");
```

Our `FormatNameValue` function would go something like this:

```
public static string FormatProperties<T>(
    this T t, string format, params string[] props)
{
 object[] values = new object[props.Length];
 for (var i = 0; i < props.Length; i++) {
  values[i] = typeof(T).GetProperty(props[i]).GetValue(t, null);
 }
 return string.Format(format, values);
}
```

This suffers from a problem common to most formatters: Once you get more than a few insertions, it becomes hard to figure out which one matches up to what. So I'm going to try something radical:

```
static Regex identifier = new Regex(@"(?<={)(.*?)(?=[:}])");
// pedants would use
//identifier = new RegEx(@"[_\p{Lu}\p{Ll}\p{Lt}\p{Lm}\p{Lo}\p{Nl}]" +
//       @"[_\p{Lu}\p{Ll}\p{Lt}\p{Lm}\p{Lo}\p{Nl}\d\p{Pc}\p{Mn}\p{Mc}]");
public static string FormatProperties<T>(this T t, string format)
{
  var values = new ArrayList();
  int count = 0;
  format = identifier.Replace(format, (m) => {
    values.Add(typeof(T).GetProperty(m.Value).GetValue(t, null));
    return (count++).ToString();
  });
  return string.Format(format, values.ToArray());
}
```

Instead of separating the properties from the format, I embed them in the format.

```
thing.FormatProperties("{Name}: {Value} (modified by {ModifiedBy})");
```

Note that I explicitly exclude colons from identifiers. That lets me do things like this:

```
var result =
  (new System.IO.FileInfo(@"C:\Windows\Explorer.exe"))
    .FormatProperties("Created on {CreationTime:F} " +
                      "{Length} bytes in size");
```

The property names are extracted and replaced with corresponding numbers, but the format string remains, allowing it to be used to alter the final formatting of the property.

Okay, at this point I figured I had gone far enough. The fun had run out, so I decided to stop.

Raymond Chen

Follow