C#

Table of content

General

Developers guide to debugging

(click on the image below)

- When dealing with objects, always check that they exist and content/elements are available

```
public IEnumerable<exSample> GetSamples(SignifyHRDAL dbContext)
{
  var samples = SampleHelper.GetSampleList();

  if (samples != null && samples.Count > 0)
  {
    //Do something
  }

  return samples;
}
```

- Format dates according to system standards (use SignifyTypeExtensions)
- Enums are singular
- Method names make sense
- Use the var keyword instead of long namespace.object.names if the type that is returned is clear from the variable initialisation.

```
public void UpdateSample()
{
  //Bad
  bool isActive = false;
  //Good
  var isActive = false;

  //Bad
  string sampleDescription = "This is the new description";
  //Good
  var sampleDescription = "This is the new description";
}
```

- Removed the old author from the method. Tracking of who made the change can be done in Git.
- Use object initializer instead of empty constructor, when setting properties.

```
public class Cat
{
    // Auto-implemented properties.
    public int Age { get; set; }
    public string Name { get; set; }

    public Cat()
    {
    }

    public Cat(string name)
    {
        this.Name = name;
    }
}
```

- Ensure that null checks are performed and handled, where necessary.
- Remember to remove unnecessary code.
- Lambda expressions – variables should make sense. Abbreviations can be used as long as it makes sense.
- Is foreach used in preference to the for(int i...) construct?
    - Foreach
    - For

- Use Translation Resources instead of hard coded text, especially for enums and meta data annotations. Also ensure that the translation resources are generated correctly using the admin page. Find steps here.
- Always prefix an interface with the letter I, for example ITransaction, IAuditable, etc.
- When naming an interface, where possible use adjective phrases, for example IRunnable, IAuditable, IPersistable, IDisposable, IComparable, IEnumerable, however, nouns can also be used such as ITransaction, IHttpModule, etc are allowed when deemed necessary.
- Use singular form in naming Enums, unless the enum represents bit-wise flags, where plural names should rather be used.
- When using generic type parameters in a generic type based classes or methods, use descriptive names such as IDictionary<TKey, TValue>, and prefix all generic type parameters with the letter T.
- Only use the letter T as a generic type parameter if it self-explanatory and usually the only generic type parameter, for example ICollection<T>, IEnumerable<T>, etc.
- When extending from the following classes, add the base class name as a suffix:
  - System.EventArgs, e.g. System.RepeaterEventArgs
  - System.Exception, e.g. System.SQLException
  - System.Delegate

Statements, expressions and operators

References

- Statements
- Expressions and operators

Additional

- Ensure the proper use of extension methods and overloads. When you have to pass in a few null values for a method, consider making another overload. Also check that the null value is handled properly to avoid null exceptions. Should this value be null to start with OR Use a POCO object
- Ensure method arguments are validated and rejected with an exception if they are invalid

Separation of concerns

The main concept here is **Single Responsibility** - "a class/method must have only one reason to change". This deals specifically with cohesion.

```
//Bad
public exSample UpdateSample(int id, string newDescription, int productId)
{
    //Method purpose is to return a sample
    var sample = SampleHelper.GetSample(id);

    //sample description is updated
    sample.description = newDescription;

  //A new product is also added the sample and the VAT calculated
  var product = ProductHelper.GetProduct(productId);
    product.VAT = product.TotalAmount * 14 / 100;
  sample.Products.Add(product);

    return samples;
}

//Good
//Method to update the sample (serves one purpose)
public exSample UpdateSample(int id, string newDescription, int productId)
{
    //Method purpose is to return a sample
    var sample = SampleHelper.GetSample(id);

    sample.description = newDescription;

    return samples;
}

//Method to add a product to the sample and do the product calculations where necessary (serves one purpose)
private exSample AddSampleProduct(exSample sample, int productId)
{
```

```
    //A new product is also added the sample and the VAT calculated
    var product = ProductHelper.GetProduct(productId);
    product.VAT = product.TotalAmount * 14 / 100;
    sample.Products.Add(product);

    return sample;
}
```

https://docs.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/architectural-principles#single-responsibility

Types

References

Build in reference types

Nullable reference types

SignifyTypeExtensions

TODO When are Types used? How are they used? Converting to other Types?

Classes and Structs

Interfaces

References

- Explicit Interface Implementation
- How to explicitly implement interface members
- How to explicitly implement members of two interfaces

Comments

References

Commenting Conventions

Arrays

References

- Arrays as Objects
- Single Dimensional Arrays
- Jagged Arrays
- Using `foreach` with Arrays
- Passing Arrays as Arguments

Generics

References

- Generic Type Parameters
  - Constraints on Type Parameters
- Generic Classes
- Generic Interfaces
- Generic Methods
- Generics and Arrays

Strings

References

- Working with Strings
- Formatting
  - Standard Numeric Format Strings
  - Custom Numeric Format Strings
  - Standard Date and Time Format Strings
  - Custom Date and Time Format Strings
  - Composite Formatting

Date Formatting (V8)

Ensure where ever a date is displayed to the user that it is done according to system setup / user preference:

```
public string CurrentDate
{
   get
   {
      return DateTime.Now.ToDateTime_Format();
   }
}
```

TODO : ToSafeString?

Any other extensions we can use/expand on

Namespaces

Usage

Make use of `using` directives to enable improved readability and limit coding effort.

```
// Not making use of using directive
namespace MyTestProgram
{
   public class MyTestClass
   {
      private void DoSomething()
      {
         if(!System.IO.Directory.Exists("C:\\TestFolder\\");
            System.IO.Directory.CreateDirectory("C:\\TestFolder\\");

         var files = System.IO.Directory.GetFiles("C:\\MainFolder\\", "*.txt");
         var fileNames = new List<string>();

         foreach (var file in files)
         {
            fileNames.Add(System.IO.Path.GetFileName(file));
         }
      }
   }
}

// Making use of using directive
using System.IO;
namespace MyTestProgram
{
   public class MyTestClass
   {
      private void DoSomething()
      {
         if(!Directory.Exists("C:\\TestFolder\\");
            Directory.CreateDirectory("C:\\TestFolder\\");
```

```
        var files = Directory.GetFiles("C:\\MainFolder\\", "*.txt");
        var fileNames = new List<string>();

        foreach (var file in files)
        {
            fileNames.Add(Path.GetFileName(file));
        }
    }
  }
}
```

Aliasing

Use aliasing to prevent ambiguous references if different namespaces have objects with the same name.

```
using DataDomain = SignifyHR.Data.Domain;
using Domain = SignifyHR.Domain;
using SignifyHR.Helpers;

namespace SignifyHR.Learning
{
  public class HaveFun
  {
    public bool IsFun(int activityId)
    {
      var activity = Domain.Activity.TryFetch(activityId);
      var rules = DataDomain.Activity.SelectActivityPathwayRuleItemsList(new SessionHelper(), activityId);

      return activity != null && rules != null;
    }
  }
}
```

Exception handling

References

Creating and Throwing Exceptions

Catch

Ensure the error is logged in a `catch`.
Actions returning a page, view or partial should return the appropriate error page.
Return the correct notification to user if elements are changed during execution or for AJAX/JSON responses.

```
// Log error
// Return correct error page/partial
public PartialViewResult PersonalDetails(int discussionId)
{
    try
    {
        return PartialView("_DiscussionPersonalDetails", new DiscussionPersonalDetailsViewModel(SessionHandler, discussionId));
    }
    catch (Exception ex)
    {
        ErrorUtilities.LogException(ex);
        return PartialView("_Error");
    }
}



// Log error
// Return correct error notification
[HttpPost]
public ActionResult Approve(int id, DiscussionSection activeTab)
{
    try
```

```
    {
        ApproveDiscussion(id, DiscussionStatus.COMPLETED);
        return Json(new { success = true, message = "Successfully approved.", redirectUrl =
Url.EncryptedAction("Index", new { id, activeTab }) });
    }
    catch (Exception ex)
    {
        ErrorUtilities.LogException(ex);
        return Json(new { success = false, message = "There was a problem saving this section." });
    }
}



// Log error
// Return correct error notification
if (itemDisplaySetting == ItemDisplaySetting.Hide)
    HidePathwayStep(ref e);
else
{
    try
    {
        var step = pwStep.Fetch(sessionHandler, Int32.Parse(hfObjectID.Value));


    }
    catch (Exception ex)
    {
        ErrorUtilities.LogException(ex);
        lblReferenceDescription.Text = string.Format("{0}{1}", " Message: ", ex.Message);
    }
}
```

When to catch

Applying exceptions are usually done for the following cases:

- Logging of error and returning an error page/message:
  - Action called by user
  - Operation required for all following processes where normal `NULL` or content checks may not be sufficient.
- Logging of error, setting error message/property and continuing with next process