This tutorial

Will teach you a basic way of doing string localization in the Unity3D engine where you will be able to switch language on the fly in run-time.

We will be coding this with C# but should not be too great a challenge translating this to you preferred scripting language.

All the strings will be saved in an XML-file meaning this might not be completely cross-platform nor the most suitable solution, something for you to research before you chose this solution for your localization needs in a live project. **I will not go into much detail about the xml-file format.**

Before we start

You need to create or open a Unity project, I will jump straight ahead into the action leaving everyone behind if you are not up to speed.

The language files

Where are strings will be saved will exist in XML-files, we will in the scope of this tutorial create two such files and put them to use.

I will create one for English and one for Swedish and name them according to the language English display name; **English.xml** and **Swedish.xml**

The content of the files will be as follows.

English.xml	**Swedish.xml**
<?xml version="1.0" encoding="utf-8" standalone="yes"?> <Root> <Phrases> <Hello>Hello World!</Hello> </Phrases> </Root>	<?xml version="1.0" encoding="utf-8" standalone="yes"?> <Root> <Phrases> <Hello>Hej Världen</Hello> </Phrases> </Root>

As you can see, we have a **Root** node, everything we do should exist inside of that and everything one level down, all the direct children of the root node will be **Categories** so we can for our own sake keep track of everything. This takes us to the important part, the actual strings that we want to localize, those exists just one level down from a category where the tag works as the **ID**, we will use that pull the text we want to show.

Now we need to save those files, and you might want to save them in the "**Resources**" folder but I will go ahead and save them in a new folder named **Languages**, that way we can have them outside of the project after building it so it is edible for everyone and you can easily install new languages.

The Code file

Is our next step, this is where we will make sure we can use our language files and retrieve our localized strings.

I start by creating the C# file and naming it to **LanguageManager** and place it inside a folder named **Scripts**.

Let us now open it up and start coding.

OK, we have our LanguageManager class, I have stripped it on everything adding to it as we go also making it inherit from **ScriptableObject** instead of **MonoBehaviour**, and for simplicity I will start by turning this class into a **Singleton** meaning we can only have one instance of it.

I achieve this by having a static member variable of the very same class and we can retrieve this instance via a public Instance property that will check if an instance exists or else create one.

I add the following code to my class.

```
using UnityEngine;

public class LanguageManager : ScriptableObject
{
   #region Singleton
   private static LanguageManager instance = null;

   public static LanguageManager Instance
   {
      get
      {
         if (instance == null)
         {
            instance = CreateInstance<LanguageManager>();
         }
         return instance;
      }
   }
   #endregion
}
```

Our next step is to create the rest of our member variables, four in total and all private:

- XmlDocument mainDoc – This will be how we access to the currently loaded xml document.
- XmlElement root – This is our root node where all the categories are listed beneath.
- string languagePath – The file path to the folder that contains our langage xml files.
- string[] languageFiles – A list of all the available file names.

However if we want to be able to use all those fancy Xml datatypes and also be able to load files we need to go to the top and type in our using directive to give us access.

Closing our Singleton region will give us the following code.

```
using UnityEngine;
using System.Xml;
using System.IO;

public class LanguageManager : ScriptableObject
{
   +Singleton

   private XmlDocument mainDoc = null;
   private XmlElement root = null;

   private string languagePath = string.Empty;
   private string[] languageFiles = null;
}
```

Let us quickly move on and override the Awake method, this method is run only once and is a good place to initialize some variables.

What we will do here is set our languagePath and also cheat a bit by calling a method we yet have to create, we will call it **Collectlanguages** and what it is supposed to do is getting all the available language files we have.

Before I show you the code we might as well create out CollectLanguages method that I will make private and in which I collect the full filename of every file in our language folder path.

```
   void Awake()
   {
      languagePath = Application.dataPath + "/Languages/";
      CollectLanguages();
   }

   private void CollectLanguages()
   {
      try
      {
         DirectoryInfo langDir = new DirectoryInfo(languagePath);
         FileInfo[] files = langDir.GetFiles("*.xml");
         languageFiles = new string[files.Length];
         int i = 0;
         foreach (FileInfo fileGo in files)
         {
            languageFiles[i] = fileGo.FullName;
            i++;
         }
      }
      catch (System.Exception e)
      {
         Debug.Log(e.Message);
      }
   }
```

As you can see I place the code in CollectLanguages inside a try-catch block, this is because I want to handle any exceptions/errors gracefully, we might want to shut down if we cannot find any files but for now I will just debug it into the log.

The next part is where we actually load the xml file, in this tutorial we will keep our file open and use it as soon as asked too, another way would to load all the string and then close the file and ask your container for the localized text instead.

We will need to methods, first one **GetLanguageFile** will retrieve the path of a language file based on the language you want.

We will step through our list of files and see if we can find the file corresponding the language we asks for.

```
private string GetLanguageFile(string language)
{
   foreach (string langGo in languageFiles)
   {
      if (langGo.EndsWith(language + ".xml"))
      {
         return langGo;
      }
   }
   return string.Empty;
}
```

We will use the above method in our **LoadLanguage** method which will open up the actual file and give us access to its sweet content.

```
public void LoadLanguage(string language)
{
   try
   {
      string loadFile = GetLanguageFile(language);
      mainDoc = new XmlDocument();
      StreamReader reader = new StreamReader(loadFile);
      mainDoc.Load(reader);
      root = mainDoc.DocumentElement;
      reader.Close();
   }
   catch (System.Exception e)
   {
      Debug.Log(e.Message);
   }
}
```

The last part of the LanguageManager code fil is to write the **Get** method, we will use that one for retrieving an actual text by sending in a category and ID, now I have placed that in a try-catch block as well because you seldom have to get a lot of text during a time critical part of the game so I have prioritized error handling over speed.

I have also made sure it replaces any new line marks with an actual new line so we can in our text insert line breaks.

```
public string Get(string path)
{
   try
   {
      XmlNode node = root.SelectSingleNode(path);
      if (node == null)
      {
         return path;
      }
      else
      {
         string value = node.InnerText;
         value = value.Replace("\\n", "\n");
         return value;
      }
   }
   catch (System.Exception e)
   {
      Debug.Log(e.Message);
      return path;
   }
}
```

Testing it

Is our next step, we will create a new C# script, I call it **Test** and we will let it type out our result in the log console. We will work solely in the Awake method.

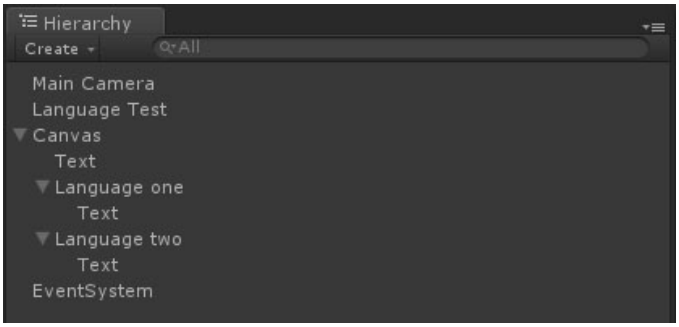
```
using UnityEngine;

public class Test : MonoBehaviour
{
   void Awake()
   {
      LanguageManager.Instance.LoadLanguage("English");
      Debug.Log(LanguageManager.Instance.Get("Phrases/Hello"));

      LanguageManager.Instance.LoadLanguage("Swedish");
      Debug.Log(LanguageManager.Instance.Get("Phrases/Hello"));
   }
}
```

We will then create a game object and place the Test component script on it and press play and as you can see if everything went as we want, it should print out “Hello World!” and “Hej V�rlden!” (The console cannot handle Swedish that is why we don’t get an ‘Ä’).

Connecting it

To Unity’s new GUI is what we want to do so the last step was just to show it worked, but to make it useful we will create a component that we can attach to a widget making it pull the text it wants.

Let’s start by creating a GUI Canvas, we will need two buttons for changing the current language and one textbox to display the localized string in.



To hook it up to the language manager we will create a C# script and place it on any Text object, in this example that is three different objects, and all named Text.

I will name the script **LocalizedText** begin with the fun part, coding.

We will only need two things, the **Start** method were we will call the method **LocalizeText** setting the text property of the text component, make sure you are including the using directive for **UnityEngine.UI**, and also a public string member called **localizedID** which we will set via the editor containing the category and ID of the string we want, for example “Phrases/Hello”.

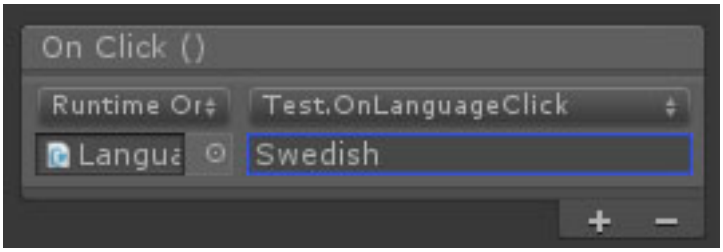
```
using UnityEngine;
using UnityEngine.UI;

public class LocalizedText : MonoBehaviour
{
   public string localizedID = string.Empty;

   void Start()
   {
      LocalizeText();
   }

   public void LocalizeText()
   {
      Text text = GetComponent<Text>();
      if (text != null)
      {
         text.text = LanguageManager.Instance.Get(localizedID);
      }
   }}
```

It is important to remember to load a language first, this is currently done in the Awake method of the Test script.

For the buttons we need to add text to the xml files

English.xml	**Swedish.xml**
<?xml version="1.0" encoding="utf-8" standalone="yes"?> <Root> <Phrases> <Hello>Hello World!</Hello> </Phrases> <LanguageNames> <English>English</English> <Swedish>Svenska</Swedish> </LanguageNames> </Root>	<?xml version="1.0" encoding="utf-8" standalone="yes"?> <Root> <Phrases> <Hello>Hej Världen!</Hello> </Phrases> <LanguageNames> <English>English</English> <Swedish>Svenska</Swedish> </LanguageNames> </Root>

The final step is to hook the buttons up, let us use the test script for this, lets add a public method called OnLanguageClick and hook up each button to use that method and send in the name of the language we want as a parameter.



In the OnLanguageClick method we add the code to change the language.

```
public void OnLanguageClick(string language)
{
   LanguageManager.Instance.LoadLanguage(language);
}
```

For the changes to actually take effect in real time we need to enforce it by calling the LocalizeText method on the LocalizedText component.

```
public void OnLanguageClick(string language)
{
   LanguageManager.Instance.LoadLanguage(language);

   LocalizedText[] texts = FindObjectsOfType<LocalizedText>();
   foreach (LocalizedText text in texts)
   {
      text.LocalizeText();
   }
}
```

Take it further

Is totally possible, one thing you can do is implement a way to have dynamic parts so that you can for instance say Hello and have it type out the players name, this I might create in a later tutorial, but for now enjoy and don't be afraid to follow us to get more tutorials on Unity3D, Unreal Engine etc… or just see what kind of shenanigans we are up to.

https://www.facebook.com/KJinteractive

http://blog.kjinteractive.net/

Kind regards

Krister Cederlund, KJ Interactive