FlexReader

Fast and lightweight CSV/XLSX documents parser.

Online Documentation

Overview

FlexReader is a mono/.net library that greatly reduces the time and effort needed to read Microsoft Excel files. It supports both `csv` (comma separated values) and `xlsx` (Excel 2007) formats. FlexReader also provides class mapping, which is able to converts raw data records directly into object collections.

Features

- Supports `csv`
- Supports `xlsx`
- Delimiter and enclose characters customization
- Cell path selector
- Span cells supported in `xlsx`
- Table manipulation, rotate, expand(rectangularize) and collapse(trim)
- Custom value converters
- Custom conversion fallback values
- Convenient mapping methods
- Generic and non-generic support
- High performance on large data records
- AOT compatible(runs on all platforms)
- Full source code and well abstracted
- Unity 3D ready(Unity tests included)
- Detailed guide and API documentation

Guide

Update

> Update from **v1.x**

If you have modified FlexReader source code, follow this guide:

1. Delete *FlexReader/SharpZip*
2. Import the latest package
3. Import Encoding support by double clicking *FlexReader/i18n.unitypackage*(optional)
4. Import Demo by double clicking *FlexReader/demo.unitypackage*(optional)

If not:

1. Delete *FlexReader*

2. Import the latest package
3. Import Encoding support by double clicking *FlexReader/i18n.unitypackage*(optional)
4. Import Demo by double clicking *FlexReader/demo.unitypackage*(optional)

Import

First import library namespace

```
using FlexFramework.Excel;
```

Document

A **Document** is a parsed object from a csv file. There are four ways to load a CSV file: path, stream, bytes and text.

Path

To load a file by its path, you should use either its absolute path or relative(to current working directory, which is the project folder) path.

Let's say we have a file under */path/to/MyProject/Assets/data/myfile.csv*. You can load it by a relative path:

```
var doc = Document.LoadAt("Assets/data/myfile.csv");
```

or by an absolute path:

```
var doc = Document.LoadAt("/path/to/MyProject/Assets/data/myfile.csv");
```

> You should NOT use this strategy on mobile/web platforms unless the path is readable.

Stream

Assume that you're getting a CSV file stream over network or any other source, you can load it directly:

```
var doc = Document.Load(stream);
```

> You probably need to close/dispose that stream after loading. Disposing is not handled internally in case you need to access that stream later.

Bytes

For a buffer array:

```
var doc = Document.Load(bytes);
```

For `www`:

```
var doc = Document.Load(www.bytes);
```

Text

Load plain text directly:

```
var doc = Document.Load(text);
```

For `WWW`:

```
var doc = Document.Load(www.text);
```

Encoding

> Default encodings are used by default. If you need custom encoding when loading, you should either use bytes or text:

```
var bytes = File.ReadAllBytes(path, Encoding.UTF8);
var doc = Document.Load(bytes);
```

```
var text = File.ReadAllText(path, Encoding.UTF8);
var doc = Document.Load(bytes);
```

Or use a `TextReader`

```
var reader = new StreamReader(stream, Encoding.UTF8);
var doc = Document.Load(reader);
```

Delimiter

A **Delimiter** is a special character to seperate each column. By default it's comma(`,`). You can overwrite that by setting the global **Delimiter**:

```
Document.Delimiter = ";";
```

That affects all the later loading process.

To restore default **Delimiter**:

```
Document.Reset();
```

Enclose

If a cell's value contains reserved characters(Delimiter and **Enclose**), it will be wrapped by **Enclose** characters. The default **Enclose** character is double quote(`"`) and can be overwritten in this way:

```
Document.Enclose = "'";
```

That affects all the later loading process.

To restore default **Enclose**:

```
Document.Reset();
```

Read

A Document is actually a table of records. In a CSV file we call each line a Row instead of record. Thus a Document is a collection of Row. And a Row is a collection of Cell.

First let's get to know about Row.

Row

To get a **Row**, you can read it by its index:

```
var row = doc[0];
```

The above code gives you the first row.

To get the count of rows in a document:

```
var count = doc.Count;
```

> The document object is readonly. To manipulate or write docs, see Manipulation.

Cell

A **Cell** is an object at a specified column, in a specified row. A cell has an Address and a **Value**. To get a cell by its index in a row:

```
var cell = row[0];
```

The above code gives you the first cell in that row. In a shorthand:

```
var cell = doc[0][0];
```

the above code gives you the first cell at the first row in that doc.

To get a cell's value(`object` type):

```
var value = cell.Value;
```

To get a cell's value in plain text:

```
var text = cell.Text;
```

To get a cell's address:

```
var address = cell.Address;
```

Get and convert a cell value

```
var value = cell.Convert<int>();
```

Or use non-generic method:

```
var value = cell.Convert(typeof(int));
```

> Convert is an extension method, not a member of Cell.

To get the count of cells in a row:

```
var count = row.Count;
```

Besides index, you can also get cells by its **Address**. Before that, let's first have a look at Address.

Address

An **Address** represents a position in an XLSX/CSV file, consisting of **Column** and **Row**. The **Row** of an **Address** is decimal. It's a one-based index rather than zero-based(which is the default strategy in most programming laguages including c#). The **Row** of an **Address** is **Bijective Hexavigesimal**, and it's also one-based.

To create an address from column and row indices:

```
var address = new Address(column, row);
```

If `column == row == 1`, you'll get `"A1"`.

You can also instantiate an address from its string representation:

```
var address = new Address("A1");
```

To convert a Bijective Hexavigesimal value to Decimal value:

```
var index = Address.ParseColumn("A");
```

The above code gives `1`.

To convert a Decimal value to Bijective Hexavigesimal value:

```
var str = Address.ParseColumn(1);
```

The above code gives `"A"`.

Addresses can be compared with each other:

```
var result = address1 > address2;
```

> `"A2"` is greater than `"A1"`. `"B1"` is greater than `"A1"`. `"A3"` is neither greater or less than `"B1"`, nor is it equal to the latter. The only condition an address being greater than another is that both of its column and row value are greater than the other's. The same applies to **less than**.

To get a cell by its address:

```
var cell = doc[address];
```

```
var cell = doc["A1"];
```

From a row:

```
var cell = doc[row];
```

```
var cell = row["A1"];
```

An exception will be thorwn when no matching cell found.

Or use extension method:

```
var cell = doc.Select("A1");
```

From a row

```
var cell = row.Select("A1");
```

> No exception will be thorwn when no matching cell found, but the result will be null.

You can select a range of cells from a doc or row. Let's get to know about Range first.

Range

Like its name, A **Range** is a range of Address. The string representation of a **Range** is like `"A1:B3"`. The left part before `:` is the starting position, the right part is end postion.

To create a **Range** from two Address objects:

```
var range = new Range(addressFrom, addressTo);
```

Or from its string representation:

```
var range = new Range("A1:B3");
```

> If the left part is **no less than** the right, an exception will be thrown.

To get the left and right part of a **Range**:

```
var left = range.From;
var right = range.To;
```

To check if an Address is between a **Range**:

```
var contains = range.Contains(address);
```

To check if an **Range** is wrapped by another **Range**:

```
var contains = outterRange.Contains(innerRange);
```

To get a range of cells from a doc:

```
var cells = doc[range];
```

Extension method:

```
var cells = doc.SelectRange("A1:A3");
```

For row:

```
var cells = row[range];
```

```
var cells = row.SelectRange("A1:A3");
```

> When getting a range of cells, the result is `IEnumerable`, it follows C#'s iterator feature. Thus the code will not be executed until the result is being enumerated, such as calling `ToArray()`.

Mapping

We've talked about converting a cell's value. But what about converting a whole row to an object, with each column corresponding to a property or field? And what about converting a doc to a collection of objects? Here comes **Mapping**.

Mapper

A **Mapper** indicates how each column corresponds to each property or field in a class or structure.

To create a **Mapper** on a class(`User` here):

	A	B	C
1	Adam	18	TRUE
2	Eve	22	FALSE

> IMPORTANT: The mapped class/structure must be marked as Serializable(`System.Serializable`)

```
[Serializable]
public class User
{
}
```

```
var mapper = new Mapper<User>();
```

Non-generic version:

```
var mapper = new Mapper(typeof(User));
```

To map the first column to one of User's property/field named `name`:

```
mapper.Map("name", 1);
```

> Remember that the column index is always one-based.

Or by Bijective Hexavigesimal representation:

```
mapper.Map("name", "A");
```

The `Map` method returns the mapper itself, letting you to call chained methods:

```
mapper.Map("name", 1).Map("age", "B").Map("gender", "C");
```

With a special multiplied mapping:

```
mapper.Map("name:1, age:B, gender:C");
```

Combine instantiation with mapping assignments:

```
var mapper = new Mapper<User>().Map("name", 1).Map("age", "B").Map("gender", "C");
```

In short:

```
var mapper = new Mapper<User>().Map("name:1, age:B, gender:C");
```

To remove an assignment:

```
mapper.Remove("name");
```

To clear all assignments:

```
mapper.Clear();
```

To copy assignments from another mapper:

```
mapper.Copy(otherMapper);
```

You can also map from a **Row** object or a collection of **Cells**, to indicate that each column's value is the property/field name, and its column index is mapped as is.

Let's say we have a row whose plain text is `"name,age,gender"`:

	A	B	C
1	name	age	gender
2	Adam	18	TRUE
3	Eve	22	FALSE

```
var row = doc[0];
mapper.Map(row);
```

The above code is equivalent to this:

```
mapper.Map("name", 1).Map("age", 2).Map("gender", 3);
```

To do a partial mapping(Assume that we only want the name and age to be mapped):

```
mapper.Map(row.SelectRange("A1:B1"));
```

The above code is equivalent to this:

```
mapper.Map("name", 1).Map("age", 2);
```

To Map a property/field to a column with default(fallback) value:

```
mapper.Map("name", 1, "Unknown").Map("age", 2, 0);
```

The above code assigns default value of `"Unknown"` to `name` property/field, and `0` to `age`. When a row is being converted, if a cell value is invalid to be assigned to a property or field(like empty cell for integer property/field), an exception will be thrown unless a default value had been given before. It's also called a **Fallback** value.

You can also enable **SafeMode** on a mapper to bypass invalid data exception. In that case, the fallback value will be `null` for reference types, and default value for value types(`0` for `int`, `false` for `bool`, etc.).

```
mapper.SafeMode = true;
```

Convert

Now we have a mapper instance. Let's convert a row with it.

```
var user = row.Convert(mapper);
```

The result will be a `User` instance if no error occurs.

You can skip the mapper creation:

```
var user = row.Convert<User>("name:1, age:B, gender:C");
```

Non-generic:

```
var user = row.Convert(typeof(User), "name:1, age:B, gender:C");
```

With row or cells:

```
var user = row.Convert<User>(row);
```

```
var user = row.Convert<User>(row.SelectRange("A1:A2"));
```

TableMapper

A **TableMapper** is just like the **Mapper** we talked before. Beyond that, it indicates which rows to be excluded from the result.

```
var mapper = new TableMapper<User>();
```

Non- generic:

```
var mapper = new TableMapper(typeof(User));
```

Map columns like we did with **Mapper**:

```
mapper.Map("name", 1).Map("age", 2).Map("gender", 3);
```

```
mapper.Map("name:1, age:B, gender:C");
```

```
mapper.Map(row);
```

```
mapper.Map(row.SelectRange("A1:A2"));
```

To exclude a row:

```
mapper.Exclude(1);
```

To exclude a bunch of rows:

```
mapper.Exclude(1, 2, 3);
```

To include a row excluded before:

```
mapper.Include(1);
```

Multiple

```
mapper.Include(1, 2, 3);
```

Table Convert

To convert each row in a doc to an object instance, we use the **TableMapper**.

```
var users = doc.Convert(mapper);
```

> when getting a range of cells, the result is `IEnumerable`, it follows C#'s iterator feature. Thus the code will not be executed until the result is being enumerated, like calling `ToArray()`.

You can also skip the mapper creation step:

```
var users = doc.Convert<User>(1);
```

Which is equivalent to:

```
var mapper = new TableMapper<User>().Map(doc[1]);
var users = doc.Convert(mapper);
```

By expression:

```
var users = doc.Convert<User>("name:1, age:B, gender:C", 1, 10);
```

Which is equivalent to:

```
var mapper = new TableMapper<User>().Map("name:1, age:B, gender:C").Exclude(1, 10);
var users = doc.Convert(mapper);
```

Attribute Mapping

Besides manually mapping docs to classes, there is an alternative:

```
[Serializable, Table(1, 2, SafeMode = true)]
public class User
{
    [Column(1)] public string name;
    [Column("B", 0)] public int age;
    [Column("C")] public bool gender;
}

var users = doc.Convert<User>();
```

Which is equivalent to

```
var mapper = new TableMapper<User>().Map("name", 1).Map("age", "B", 0).Map("gender",
"C").Exclude(1, 2);
mapper.SafeMode = true;
var users = doc.Convert(mapper);
```

Manipulation

There are three extension methods to manipulate a table: **Expand**, **Collapse** and **Rotate**. The source table itself won't be modified. Instead, a deep cloned table will be created, modified and returned. Cell addresses will be recalculated after that.

Expand

If a table gets expanded, all length of all rows will be the same, i.e. empty cells will be appended to shorter rows(rows with less cells).

```
var expandedDoc = doc.Expand();
```

Collapse

As opposed to **Expand**, empty cells at each row's boundary will be removed. Note that if there is a series of empty cells at the boundary, all of them will be removed. If a boundary row consists of empty cells, it will be removed. The same applies to any row with no cells.

```
var collapsedDoc = doc.Collapse();
```

Rotate

This method rotates a table by 90 degrees clockwise. The table will first be **expanded** since a non-square table cannot be rotated.

```
var rotatedDoc = doc.Rotate();
```

Recalculate

When this method gets called, all cell's address will be recalculated based on their current position. Note that this method modifies the table and has no return value instead of a clone.

```
doc.Recalculate();
```

Converters

Converters are important because it tells how plain texts are parsed to specified data types. Converters must be created and registered to `ValueConverter`. To create a converter, there are two ways.

Converter Class

Here is a sample converter class, which is already a built-in converter of `Vector3` type.

```
public sealed class Vector3Converter : CustomConverter<Vector3>
{
  public override Vector3 Convert(string input)
  {
    if (!Regex.IsMatch(input, @"^\([-+]?[0-9]*\.?[0-9]+\b,[-+]?[0-9]*\.?[0-9]+\b,[-+]?[0-9]*\.?[0-9]+\b\)$"))
    {
      throw new FormatException("Vector3 value expression invalid: " + input);
    }

    string[] parameters = Split(input.Trim('(', ')'), ',');
    float x = ValueConverter.Convert<float>(parameters[0]);
    float y = ValueConverter.Convert<float>(parameters[1]);
    float z = ValueConverter.Convert<float>(parameters[2]);
    return new Vector3(x, y, z);
  }
}
```

Don't forget to register it:

```
ValueConverter.Register<Vector3Converter>();
```

Then it'll be applied automatically whenever you try to convert to `Vector3` type:

```
var value = ValueConverter.Convert<Vector3>("(3,2,1)");
```

```
var value = cell.Convert<Vector3>();
```

To unregister that:

```
var successed = ValueConverter.Unregister<Vector3>();
```

> For a specific data type, only one converter can exist. Any newly registered one overwrites the existing one.

Converter Delegate

This allows you to register a delegate method as a converter. Let's rewrite the above one:

```
ValueConverter.Register(input =>
{
  if (!Regex.IsMatch(input, @"^\([-+]?[0-9]*\.?[0-9]+\b,[-+]?[0-9]*\.?[0-9]+\b,[-+]?[0-
9]*\.?[0-9]+\b\)$"))
  {
    throw new FormatException("Vector3 value expression invalid: " + input);
  }

  string[] parameters = Split(input.Trim('(', ')'), ',');
  float x = ValueConverter.Convert<float>(parameters[0]);
  float y = ValueConverter.Convert<float>(parameters[1]);
  float z = ValueConverter.Convert<float>(parameters[2]);
  return new Vector3(x, y, z);
});
```

Converters for the following types are pre-registered:

- `Boolean`
- `Byte`
- `Char`
- `DateTime`
- `Decimal`
- `Double`
- `Int16`
- `Int32`
- `Int64`
- `SByte`
- `Single`
- `String`
- `UInt16`
- `UInt32`
- `UInt64`

And special types:

- `Color`
- `Color32`
- `Rect`
- `Vector4`
- `Vector3`
- `Vector2`
- `Object`
- `string[]`
- `int[]`
- `float[]`

- `List<string>`
- `List<int>`
- `List<float>`
- `Dictionary<string, string>`
- `Dictionary<string, int>`
- `Dictionary<int, int>`
- `Dictionary<int, string>`

You can also overwrite default converters.

```
ValueConverter.Register(i => i.ToUpperCase());
```

The above string converter capitalizes all string values.

```
ValueConverter.Register(i => Int32.Parse(i) * 2);
```

The above int converter doubles the result.

To removed all converters:

```
ValueConverter.Empty();
```

To restore default converters:

```
ValueConverter.Reset();
```

Excel 2007

Unlike **CSV**, an Excel 2007 file(.xlsx) can store multiple docs. We call them **WorkSheet**. And we call the XLSX file **WorkBook**.

WorkBook

To load a **WorkBook**:

```
var book = new WorkBook("path/to/file.xlsx");
```

```
var book = new WorkBook(bytes);
```

```
var book = new WorkBook(stream);
```

Get the count of **WorkSheet** in a book:

```
var count = book.Count;
```

WorkSheet

A **WorkSheet** is just like a **Document** we talked before.

Get a **WorkSheet** in a **WorkBook** by index:

```
var sheet = book[0];
```

By name:

```
var sheet = book["sheet1"];
```

Get the name of a sheet:

```
var name = sheet.Name;
```

Get the ID of a sheet:

```
var id = sheet.ID;
```

Get the count of **Cells** in a sheet:

```
var count = sheet.Count;
```

Cell

Unlike cells in a CSV **Document**, cells form **WorkSheet** are typed. Their **Value** might not be string. Possible value types are:

- `String`
- `Boolean`
- `Int32`
- `Int64`
- `Single`
- `Double`

Check if a cell is of `Boolean` type:

```
var result = cell.IsBoolean;
```

Get the `Boolean` value:

```
var value = cell.Boolean;
```

> If that cell's value is not of `Boolean` type but you called its `Boolean` value, an exception will be thrown.

The above code does not convert the cell's value.

Span

A cell in a **WorkSheet** might be a span, which is a merged part and has no value(**Value** will be `null`).

To check if a cell is a span:

```
var result = cell.IsSpan;
```

If you call `Merge()` on a **WorkBook** or **WorkSheet**, values will be assigned to spans. Span cells of the same merged part shares the same value.

```
book.Merge();
```

The above code merges all sheets in that book. To merge a specific sheet only:

```
sheet.Merge();
```

Write

Row, **Document** are readonly. But there are workarounds: **Row.Cells** and **Document.Rows**. They expose the inner collection of **Row** and **Document**. You can **Add**, **Remove** and do other tricks on them like any other `IList<T>` items.

Loading

This guide demonstrates commonly used csv/xlsx loading strategies.

Scene Reference

Using this strategy, it is required to rename xlsx extension to `.bytes` to make them serializable.

e.g.

`data.xlsx` --> `data.xlsx.bytes`

Put csv/xlsx files into your project folder, add `TextAsset` fileds to your script, and assign them file assets in inspector view.

```
using UnityEngine;
using FlexFramework.Excel;

public class Test : MonoBehaviour
{
    public TextAsset csv;
    public TextAsset xlsx;

    public void LoadCSV()
    {
        var doc = Document.Load(csv.text);
    }
```

```
    public void LoadExcel()
    {
        var book = new WorkBook(xlsx.bytes);
    }
}
```

Resources Folder

Using this strategy, it is required to rename xlsx extension to `.bytes` to make them serializable.

e.g.

`data.xlsx` --> `data.xlsx.bytes`

Put csv/xlsx files into any *Resources* folder, load them without file extensions.

```
using UnityEngine;
using FlexFramework.Excel;

public class Test : MonoBehaviour
{
    public void LoadCSV()
    {
        // Resources/my-data.csv
        var asset = Resources.Load<TextAsset>("my-data");
        var doc = Document.Load(asset.text);
    }

    public void LoadExcel()
    {
        // Resources/my-data.xlsx.bytes
        var asset = Resources.Load<TextAsset>("my-data.xlsx");
        var book = new WorkBook(asset.bytes);
    }
}
```

StreamingAssets

This strategy implements asynchronous operations since it's recommended to load resources from *StreamingAssets* folder with `UnityWebRequest` due to the fact that *StreamingAssets* is compressed on some platforms(Android, WebGL).

e.g.

Put csv/xlsx files into *Assets/StreamingAssets* folder, load them with coroutines.

```
using System.IO;
using System.Collections;
using UnityEngine;
using UnityEngine.Networking;
using FlexFramework.Excel;
```

```
public class Test : MonoBehaviour
{
    public IEnumerator LoadCSV()
    {
        // StreamingAssets/my-data.csv
        var url = Path.Combine(Application.streamingAssetsPath, "my-data.csv");
        using (var req = UnityWebRequest.Get(url))
        {
            yield return req.SendWebRequest();
            var bytes = req.downloadHandler.data;
            var doc = Document.Load(bytes);
        }
    }

    public IEnumerator LoadExcel()
    {
        // StreamingAssets/my-data.xlsx
        var url = Path.Combine(Application.streamingAssetsPath, "my-data.xlsx");
        using (var req = UnityWebRequest.Get(url))
        {
            yield return req.SendWebRequest();
            var bytes = req.downloadHandler.data;
            var book = new WorkBook(bytes);
        }
    }
}
```

Readable Path

This strategy requires a readable path(e.g. `Application.dataPath`, `Application.persistentDataPath`) to work with.

e.g.

Put csv/xlsx files into *Assets/MyDocs* folder. This only works in Editor mode. In standalone/mobile mode, you should ensure file path exists either by copying files over or downloading and saving them.

```
using System.IO;
using UnityEngine;
using FlexFramework.Excel;

public class Test : MonoBehaviour
{
    public void LoadCSV()
    {
        // MyDocs/my-data.csv
        var path = Path.Combine(Application.dataPath, "MyDocs", "data-1.csv");
        var doc = Document.LoadAt(path);
    }

    public void LoadExcel()
    {
```

```
        // MyDocs/my-data.xlsx
        var path = Path.Combine(Application.dataPath, "MyDocs", "data-1.xlsx");
        var book = new WorkBook(path);
    }
}
```

Internet

This strategy implements asynchronous operations.

e.g.

Download csv/xlsx files on and load them on the fly over http requests.

```
using System.IO;
using System.Collections;
using UnityEngine;
using UnityEngine.Networking;
using FlexFramework.Excel;

public class Test : MonoBehaviour
{
    public IEnumerator LoadCSV()
    {
        var url = "https://mysite.com/my-data.csv";
        using (var req = UnityWebRequest.Get(url))
        {
            yield return req.SendWebRequest();
            var bytes = req.downloadHandler.data;
            var doc = Document.Load(bytes);
        }
    }

    public IEnumerator LoadExcel()
    {
        var url = "https://mysite.com/my-data.xlsx";
        using (var req = UnityWebRequest.Get(url))
        {
            yield return req.SendWebRequest();
            var bytes = req.downloadHandler.data;
            var book = new WorkBook(bytes);
        }
    }
}
```

Reference

https://docs.unity3d.com/Manual/Coroutines.html

https://docs.unity3d.com/Manual/UnityWebRequest.html

https://docs.unity3d.com/ScriptReference/Application-persistentDataPath.html

https://docs.unity3d.com/Manual/StreamingAssets.html

Encoding

If you encounter this Exception:

> NotSupportedException: CodePage XXX not supported...

It means you're missing some encoding support. This usually happends at runtime but not in Editor mode. The solution is to import **CodePage.unitypackage** that shipped with FlexReader.

What's in the package:

- I18N.dll
- I18N.CJK.dll
- I18N.MidEast.dll
- I18N.Other.dll
- I18N.Rare.dll
- I18N.West.dll
- link.xml

You can keep only some of them based on your needs.

I18N.dll

Internationalization base lib. It's required to use other I18N modules.

I18N.xxx.dll

Quick pick:

- For Western/European encodings, use **I18N.West.dll**
- For Asian/Chinese,Japanese,Korean encodings, use **I18N.CJK.dll**
- For Arabic/MidEast encodings, use **I18N.MidEast.dll**

You may want to check by CodePage mentioned in that exception:

CodePage	Requires	CodePage	Requires
51932	I18N.CJK	932	I18N.CJK
936	I18N.CJK	949	I18N.CJK
950	I18N.CJK	-	-
10000	I18N.West	10079	I18N.West
1250	I18N.West	1252	I18N.West
1253	I18N.West	28592	I18N.West
28593	I18N.West	28597	I18N.West
28605	I18N.West	437	I18N.West
850	I18N.West	860	I18N.West
861	I18N.West	863	I18N.West
865	I18N.West	-	-
1254	I18N.MidEast	1255	I18N.MidEast
1256	I18N.MidEast	28596	I18N.MidEast
28598	I18N.MidEast	28599	I18N.MidEast
38598	I18N.MidEast	-	-
1251	I18N.Other	1257	I18N.Other
1258	I18N.Other	20866	I18N.Other
21866	I18N.Other	28594	I18N.Other
28595	I18N.Other	57002	I18N.Other
874	I18N.Other	-	-
1026	I18N.Rare	1047	I18N.Rare
1140	I18N.Rare	1141	I18N.Rare
1142	I18N.Rare	1143	I18N.Rare
1144	I18N.Rare	1145	I18N.Rare
1146	I18N.Rare	1147	I18N.Rare
1148	I18N.Rare	1149	I18N.Rare
20273	I18N.Rare	20277	I18N.Rare
20278	I18N.Rare	20280	I18N.Rare

CodePage	Requires	CodePage	Requires
20284	I18N.Rare	20285	I18N.Rare
20290	I18N.Rare	20297	I18N.Rare
20420	I18N.Rare	20424	I18N.Rare
20871	I18N.Rare	21025	I18N.Rare
37	I18N.Rare	500	I18N.Rare
708	I18N.Rare	737	I18N.Rare
775	I18N.Rare	852	I18N.Rare
855	I18N.Rare	857	I18N.Rare
858	I18N.Rare	862	I18N.Rare
864	I18N.Rare	866	I18N.Rare
869	I18N.Rare	870	I18N.Rare
875	I18N.Rare	-	-

link.xml

This prevents code stripping. Remove unused assemblies to reduce build size.

```
<linker>
    <assembly fullname="I18N" preserve="all"/>
    <assembly fullname="I18N.West" preserve="all"/>
    <assembly fullname="I18N.CJK" preserve="all"/>
    <assembly fullname="I18N.MidEast" preserve="all"/>
    <assembly fullname="I18N.Other" preserve="all"/>
    <assembly fullname="I18N.Rare" preserve="all"/>
</linker>
```

Reference:

https://github.com/mono/mono/tree/master/mcs/class/I18N

https://www.science.co.il/language/Locale-codes.php

Examples

Load

Stream

> Load a csv document from stream

```
using(var stream = File.OpenRead("path/to/file.csv"))
{
    Document doc = Document.Load(stream);
}
```

Bytes

> Load from bytes

```
Document doc = Document.Load(File.ReadAllBytes("path/to/file.csv"));
```

Plain String

> Load from string

```
Document doc = Document.Load("a,b,c\n1,2,3\n4,5,6\n7,8,9");
```

This loads a table:

a	b	c
1	2	3
4	5	6
7	8	9

Custom Delimiter

> Load with custom delimiter

```
Document.Delimiter = ';';
Document doc = Document.Load("a;b;c\n1;2;3\n4;5;6\n7;8;9");
Document.Reset();
```

Dump

Basic

> Dump the table back to string

```
string plain = doc.Dump();
```

Customize

> Dump the table with custom delimiter and enclose characters

```
string plain doc.Dump(';', '\"');
```

which returns `a;b;c\n1;2;3\n4;5;6\n7;8;9`

Read

Basic

> Access a row

```
Row row = doc[0];
```

> Access a cell

```
Cell cell1 = row[0];
Cell cell2 = doc[0][0];
```

> Access by selectors

```
Cell cell1 = table.Select("A1");
Cell cell2 = table["B1"];
Cell cell3 = row.Select("B2");
Cell cell4 = row["C3"];
```

> Select a range of cells

```
Cell[] cells = table.SelectRange("A1:C3").ToArray();
```

Cell

> Properties

```
Address address = cell.Address; // C2
int column = address.Column; // 3
int row = address.Row; // 2
object value = cell.Value;
string text = cell.Text;
```

> Convert Value

```
int value1 = cell1.Convert<int>();
int value2 = (int)cell2.Convert(typeof(int));
```

> More value type

```
Color color = cell1.Convert<Color>();
Vector3 vector = cell2.Convert<Vector3>();
Rect rect = cell3.Convert<Rect>();
int[] array = cell4.Convert<int[]>();
List<Vector3> list = cell5.Convert<List<Vector3>>();
```

Convert

Class Mapping

> Using attributes

```
[Serializable]
class User
{
    public string name;
    public bool member;
    public int age;
    public string phone;
}
```

```
var mapper = new Mapper<User>().Map("name", 1).Map("member", "B").Map("age",
3).Map("phone", "D");
User user = row.Convert(mapper);
```

or

```
User user = row.Convert<User>("name:1, member:B, age:C, phone:4");
```

or

```
Row index = doc[0];
User = row.Convert<User>(index);
```

Fallback values

```
// when column value format is invalid, fallback value will be used, rather than an
exception being thrown
var mapper = new Mapper<User>().Map("name", 1, "Unknown").Map("member", "B",
false).Map("age", 3, 0).Map("phone", "D", "-");
User user = row.Convert(mapper);
```

Safe Mode

```
var mapper = new Mapper<User>().Map("name:1, member:B, age:C, phone:4");
// no FormatException will be thrown in safe mode
mapper.SafeMode = true;
User user = row.Convert(mapper);
```

Table mapping

> Basic

```
// mapping and skip row 1
var mapper = new TableMapper<User>().Map("name:1, member:2, age:3, phone:4").Exclude(1);
Users[] users = doc.Convert<user>(mapper);
```

or

```
// use row 1 as index row
Users[] users = doc.Convert<user>(1);
```

> Using attributes

```
[Serializable, Table(1, 2)] // skip the first and second rows
class User
{
    [Column(1)] public string name;
    [Column("B", true)] public bool member; // fallback value is true
    [Column(3, 18)] public int age; // fallback value is 18
    [Column("D")] public string phone;
}
```

```
Users[] users = doc.Convert<user>();
```

Custom converters

> Register custom converters for new data types or to override existing converters

```
// converter for int value
ValueConverter.Register(input => Int32.Parse(input) * 100);

// converter for string value
ValueConverter.Register(input => input.ToUpperCase());

// converter for Point value
ValueConverter.Register(input => {
    //...
    return new Point();
})
```

Modify

Manipulation

> Add/remove cells/rows

```
var row = new Row();
row.Cells.Add(cell);
doc.Rows.Add(row);
```

Clone

```
var clone = doc.DeepClone();
var rowClone = row.ShallowClone();
```

Rotate

```
var rotated = doc.Rotate();
```

a	b	c
1	2	3
4	5	6
7	8	9

becomes

7	4	1	a
8	5	2	b
9	6	3	c

XLSX

Load

```
using(var stream = File.OpenRead("path/to/file.xlsx"))
{
    WorkBook doc = new WorkBook(stream);
}
```

or

```
WorkBook doc = new WorkBook(File.ReadAllBytes("path/to/file.xlsx"));
```

or

```
WorkBook doc = new WorkBook("path/to/file.xlsx");
```

WorkSheet

```
WorkSheet sheet1 = doc[0];
WorkSheet sheet2 = doc["sheet2"];
string id = sheet1.ID; // 1
string Name = sheet1.Name; // sheet1
ReadOnlyCollection<Range> spans = sheet1.Spans;
```

Selector

```
Cell cell1 = doc.Select("sheet1!A1");
Cell cell2 = doc["sheet1"]["A1"];
Cell cell3 = sheet1[0][1];
Cell cell4 = sheet1["A2"];
Cell cell5 = sheet1.Select("A2");
Cell[] cells = doc.SelectRange("sheet1!A1:C3").ToArray();
```

Typed Cell

Unlike CSV cells which are plain strings, cells in XLSX worksheets are typed. Their underlying values are stored as the type of `object`, which can be accessed by the property `Cell.Value`. The value type might be `string`, `int`, `float`, `boolean` or `null`. Casted values can be fetched by following convenient properties.

> This does not convert value type, and should be used with xlsx only

```
// equivalent to (cell.Value != null && cell.Value.GetType() == typeof(int))
if(cell.IsInteger)
{
    int value = cell.Integer;
    // equivalent to (int)cell.Value
}
if(cell.IsBoolean)
{
    int value = cell.Boolean;
}
if(cell.IsSingle)
{
    int value = cell.Single;
}
```

Span Cell

> Span cell will have a value of null until being merged

```
if(cell.IsSpan)
{
    object value = cell.Value; // null
}
doc.Merge(); // merge all sheets
sheet1.Merge(); // merge this sheet only
if(!cell.IsSpan) // false
{
    object value = cell.Value; // not null
}
```