# Record, Dictionary & Tuple Types

Our first task in building the flag image comment will be in dealing with `RegionInfo` similarly to before, however, this time, we need more than the region code - we also want the English and Native names. Our region list therefore needs to be extended to include this information. In object-oriented programming one would be inclined to create a class representing a region that includes properties that maintain the code, English and Native names. In F# one can certainly do this however, without getting into technicalities of memory management and garbage collection[20] of certain types of objects, we think it better not to use the class methodology in this instance.

We are going to initially use a "Dictionary" to store our region information. In .NET a dictionary is defined as a collection of strongly-typed key-value pairs. We know what a collection is - for example our previous lists represented a collection of elements and we used for to iterate over a collection of `CultureInfo` objects returned by the `GetCultures` method of the `CultureInfo` class. As to a key-value pair this just means that each value in the collection is "keyed", that is reference-able, via a key. For example, the value of the English name of South Africa for a region is keyed by the region code, `ZA`. So, the key-value pair is then `"ZA", "South Africa"`.

That pair, by itself, doesn't satisfy our requirement - we'd like to include the Native name so our key-value pair would have to be "extended" to something like `"ZA", ("South Africa", "iNingizimu Afrika")`. When we were just using the English name the key-value pair had `type string` for the key and `string` for the value - should we concatenate the English and Native names and say that's a single `string` as in, for example, `"South Africa, iNingizimu Afrika"`? You can do this and thereby retain `string` as the type for both the key and the value but that's not very constructive in terms of our exercise; we'll "get there" but we have to do some work first!

Dictionaries are strongly-typed because they permit you to declare what the type of the key and value are - they don't have to be string and they don't have to be the same. Imagine, for example, if we had a class, say called `RegionData`, with these properties and the class also had in integer which was the regions' `GeoId`, we could then define a dictionary with the key-value type pair `int, RegionData`. You could then access the `RegionData` properties for a specific `GeoId` instance by using the `GeoId` integer as a "key" to look-up the "values" of the properties in the class instance of `RegionData` maintained in the dictionary for *that* key.

## Record Types

As we've said, in F# we'd rather not define a class in this case - we're going to use something called an F# Record Type that has nothing to do with the .NET Framework, it is a type defined only in the F# language. Record types are more like C++ structures than classes however, for .NET compatibility, F# also supports structures (which may optionally be highlighted as such by using the attribute `[<Struct>]`) so structures are not record types, nor vice-versa - they behave differently in terms of their use of system resources despite any syntactical similarities. Furthermore, the underlying mechanisms for the way that F# creates, handles and maintains record types and classes is very, very different. Unless you know the difference between a "heap" and a "stack" and their implications upon the use of system resources just bear in mind, for the moment, that...

> If you are creating many instances of an entity as, for example, in populating a Region Dictionary, it is "better" to use a record type rather than a class.

Declaration of a record type, unlike what you've seen to date regarding classes and their members, is actually very straightforward. Let's create a new script file; in your `saTrilogy.Resources` project create a new script called **RegionNames.fsx** and place it in your **fsx** folder in the **Solution Explorer**. Again, it's useful to include the FsEye snippet. If you re-execute the FsEye instantiation code it will clear the existing contents of the FsEye window, however, you may wish to manually clear the contents and a reset of your FSI session will accomplish that or, in FsEye, just use the **Clear All** button but I often prefer to "kill" everything and start again, "from scratch".

We have a requirement to encode a string for the English name and a string for the Native name so we can define a corresponding record type as follows...

```
type RegionNames =
  { English : string
    Native : string }
```

Firstly, note the usage of braces {} and don't confuse this with what we've seen of sequences so far. Other than that...

- You use, once again, the `type` keyword, followed by a name for the record type and then an = to start the declaration of the record type body content which is encapsulated by a pair of braces {}
- We put each element of the record type on a separate line so we don't need to use a semi-colon to delimit the elements. You can use semi-colons between the elements if you wish - as you must if not declared on separate lines. If you get into any trouble then do use a semi-colon after each element noting that, in many circumstances the F# Power Tools code formatter (with a keyboard shortcut of **Ctrl+K,Ctrl+D**) may remove the semi-colons.

  For each element:

  - We give it a name - we've used `English` and `Native`.
  - You specify the `type` which follows, as usual, the colon. Both of our elements are of `type string`.

By default, a record type is immutable as is each named element of its structure. This suits our purposes as in this case, we're not going to want to change the values once they're assigned. In the future though, we will see a method whereby we can declare a named element to be mutable if so desired. Mutability is a subject much discussed hereafter.

**Video**

27. F# Record Type - Introduction

I've not added the region code to the record type since I'm going to use that as a "key" in a dictionary's key-value pair to reference the element values of this record type for the specified key. So, as in the foregoing discussion of key-value pair, our dictionary pair would be typed as `<string, RegionNames>`.

## Dictionaries

As we've mentioned immutable - insofar as it being a default characteristic of a record type and its elements, we should now point out that .NET Dictionaries are, by default, mutable; one can add and remove key-value pairs and one can modify a keyed elements value. This is not, generally, what we like to deal with in F# and, certainly in compiling a dictionary of region information, there's no need for either the record type, or its contents to change once defined. We

will be populating our region information from the `System.Globalization.CultureInfo` class and *its* contents won't alter unless your operating system is updated. To get around this "limitation" in .NET Dictionaries, F# defines its own dictionary object using the keyword `dict`. A `dict` is, essentially, an immutable .NET Dictionary. "Unfortunately", the F# `dict` has an `Add` method that makes it appear mutable. However, all `Add` does is create a *new* `dict` from the old and new key-value pairs and, especially if you need to `Add` a lot of key-value pairs, this strikes me as a potentially wasteful and costly exercise.

Firstly, let's add an `open System` and an `open System.Globalization` in our script, then, for our purposes, we declare our dictionary as follows...

```
let Regions =
  dict [ for culture in
              CultureInfo.GetCultures
                (CultureTypes.SpecificCultures) do
            let ri = new RegionInfo(culture.LCID)
            yield
              ri.TwoLetterISORegionName,
              { English = ri.EnglishName
                Native = ri.NativeName } ]
```

This actually looks familiar; as before...

- We use an enumeration over the `SpecificCultures` filtered `GetCultures` and declare an instance of the `RegionInfo` class against the enumerated items' locale identifier.
- We then `yield` "something"

Without the keyword `dict`, this looks just like our list comprehension only that we yield something different! It is, in fact, a list comprehension and you could test that by running the statement if FSI having deleted the keyword `dict` - after, of course, pre-executing the `open` statements and the declaration of the record type!

In this case, if you execute the entire script, the FSI output is more instructive in terms of it being immediately (relatively!) obvious about how the value type "looks". After execution, type `Regions;;` into the FSI output window and you should see something resembling the following - since I've edited the output to highlight the structure of the value...

```
> Regions;;
val it :
  Collections.Generic.IDictionary<string,RegionNames> =
    seq [
      [DJ, FSI_0006+RegionNames]
```

```
6             { Key = "DJ";
                Value = {English = "Djibouti"; Native =
...   "Yabuuti";};};
8           [ZA, FSI_0006+RegionNames]
              { Key = "ZA";
10              Value = {English = "South Africa"; Native =
...   "i-South Africa";};};
            [ET, FSI_0006+RegionNames]
12            { Key = "ET";
                Value = {English = "Ethiopia"; Native =
...   "□□□□□□";};};
14          [AE, FSI_0006+RegionNames]
              { Key = "AE";
16              Value = {English = "United Arab Emirates";
...   Native = □□□□□□□□□" □□□□□□□ "□□□□□□□□;};}; ...]
```

Note, firstly, the type of the value type regions - it is `Collections.Generic.IDictionary<string,RegionNames>`. The `I` in front of `Dictionary` means, in .NET terms, that we're addressing an *interface* - we can "ignore" such niceties for the time being - just regard it as a Dictionary - but bear in mind it's declared as an F# `dict` and that makes it immutable. You'll note the `<>` angle brackets suffixing the `IDictionary`; previously we just had the likes of `list<int>` but, because a dictionary is made up of a key-value pair, we have a pair of types - the `type` of the key is `string` and that of the value is `RegionNames`. Remember this distinction - life will get a bit more confusing when we come to looking at tuples!

After the `val` statement in the output you'll notice the keyword `seq` and it's followed, on the next line, by `[`, then, at the end of the `seq` there is a closing `]` after the ellipsis. The `ToString` representation of the sequence is curtailed after showing only the first four elements in the sequence . Now, I find this objectionable and annoying...

- `seq` is the `type` that identifies a sequence and, in all we've said to date, when we look at sequences they're generally identifiable because they're encapsulated by curly braces - `{}`. Here we have `[]`.
- A `dict` is not only immutable, it is definitely not lazy - you can see that the comprehension is evaluated, in full, at the time of declaration and, in FsEye, there is a node under the Regions node entitled `ICollection` that tells you there are 142 elements in the collection - even without expanding the node! Yet we know, even though we haven't yet looked at sequences in detail, that sequences *are* lazy!

It seems to me that this syntax is a botched up mess arising from the fact that `dict` is using an underlying .NET Dictionary and that "everything" in .NET, collection-wise (any .NET entity that implements the `IEnumerable` interface), presents itself as a `seq` through it's damnable `GetEnumerator` method using

the `IEnumerable` interface. My conclusion is that `dict` is not, actually, a native F# type and, as such, we'll get inconsistencies like this that we just have to live with - as is usually the case when dealing with the .NET Framework from F#.

That said, let's consider an element, a key-value pair instance, of our dictionary - take the one for South Africa...

```
[ZA, FSI_0006+RegionNames]
{ Key = "ZA";
  Value =
    { English = "South Africa";
      Native = "i-South Africa";
    };
};
```

**Video**

28. F# dict - Introduction.

Firstly, the key-value pair is designated by the encapsulating brackets `[]`. The key and value are comma-delimited so the *key* is `ZA` and the *value* is `FSI_0006+RegionNames`! Well, not strictly true; F# can't express the value as though it were a simple type such as a string for the key or, say an integer, so it instead appears to enter a "place-holder" which is what we see. In this case, the place-holder is the internally generated FSI module name that comprises your compiled script code (the prefix `FSI_0006`) concatenated using a `+` sign with the name of the `type` comprising the dictionary value - the `RegionNames` suffix. The output may as well be in Double Dutch - it's not very helpful to most people! Consequently, FSI is also showing, following the key-value pair, the `ToString` representation of the key and value. For the former it's a simple string, for the latter it displays the record type field names and their associated value. All in all, far more readable and understandable. Notice how the record type field values are semi-colon delimited - as is the key-value pair although, for a key-value pair you would never, in code, use the semi-colon as a delimiter - always use a comma.

Sometimes, deciphering FSI output is hard and, if necessary, it helps to copy the output into, for example, Notepad++ and then mess around with the formatting and line-breaks - pretty much as I have done in the foregoing listings!

It's "interesting", from the script code point of view, that we didn't explicitly tell F# that we were wanting to yield an instance of our `RegionNames` record type - F# figured this out for itself and, if it can't, it'll let you know about it before you compile your code. You can see that in the `yield` expression we specify the required key value - here as `ri.TwoLetterISORegionName`, we then specify a comma to delimit the key from the value for the yielded dictionary element. For the value part, we "mimic" our record type declaration but instead of specifying `: type` after the field name, we use `= value` to assign a value to the field. It's this "full" specification of element/field names and assignments that permits F# to impute what record type you're wanting to create an instance of.

## Uniqueness of a Dictionary Key

In FsEye you'll note that `Regions` has 142 elements - key-value pairs. It'll take you a moment but expand the `GetEnumerator` node and count, visually, how many key occurrences there are for the key `ZA` - South Africa. If you come up with an answer other than one you need to look again or have your eyesight tested!

Add the following code to your script...

```
let zaDictKeys =
  [ for key in Regions.Keys do
      if key = "ZA" then
        yield Regions.[key] ]
```

Here we're just creating a list by enumerating all keys of our dictionary and only yielding the dictionary value where the key name equals the string `ZA`. Execute this code to verify that there is, indeed, only a single `ZA` key in the `Regions` dictionary. Now add this code to your script...

```
let RegionList =
  [ for culture in
        CultureInfo.GetCultures
          (CultureTypes.SpecificCultures) do
      let ri = new RegionInfo(culture.LCID)
      if ri.Name = "ZA" then
        yield
          ri.TwoLetterISORegionName,
          { English = ri.EnglishName
            Native = ri.NativeName } ]
```

We're using the same enumeration as for the `Regions dict` but outputting a list and just filtering the `yield` by only taking regions whose Name is `ZA`. Execute this code and FsEye will show you that the list has nine elements! Clearly, the region Name property does not provide uniqueness in and of itself for the region information and, as you can see, in the `Regions` dictionary we've only "pulled" that last element shown in our list. Apart from the fact that it's clear that the key of a dictionary must be a unique value it's also clear that if you "feed" a dictionary via a comprehension, you won't get any notification that there are duplicates in potential key values - it just takes the last one it come across from the source of the "feed".

Why do we care? Recall, our objective is to form a comment for a region flag that comprises of the English and Native names. You can see that in FsEye, for the list `RegionList`, that whilst the region with name `ZA` has only one English name, there are at least five distinct Native names and we'll need these for our RESX flag comment. Furthermore, it's clear that if our key comprised of both

the region name, `ZA` and the Native name, we'd still be "dropping" values from the comprehension in creating a dictionary with such a "compound" key, but, at least we'll have extracted the distinct Native names, as required.

It therefore appears that we may have introduced a real burden (instructional, nonetheless) in our requirement of showing the Native name in a resource image comment - which Native name or all of them? So we need to take steps to eliminate this indeterminacy in Native name. We'll stick with `ZA` as our working "sample" since it has the merit of using the Latin alphabet throughout - if you want to get really confused run the `RegionList` code again after having changed the region name to `IN` for India for `CN` for China!

**Video**

29. Uniqueness of a dictionary key.

One final point to note here that re-iterates the "ordered" characteristic of a list is that you'll see that even though our dictionary keys are unique because of the characteristics of being a dictionary key, the keys are neither displayed nor stored in a "sorted" order - which can make looking though keys, visually, tiresome.

## Linq & IEnumerable Collections

A lot of the "practical" collections we'll want to deal with are sourced from the .NET Framework - for example, `GetCultures`. Of course, we'll be creating native F# collections and you'll find, ultimately, that dealing with these is pretty straightforward. The overriding feature of .NET collections, as we've seen, is that they present as a .NET collection via the `IEnumerable` interface which, from F# will be treated as a sequence. Even then, in some circumstances, the `IEnumerable`, as presented, is not something one can "handle" effectively, immediately in F# since the `IEnumerable` is returned as a collection of objects - the .NET `object` type so, when coding your application, Intellisense won't be able to help you with what methods/properties are available. In dealing with `GetCultures` so far we've been fortunate as this presents as a strongly typed collection of `CultureInfo`. Later on, we won't be so fortunate.

Before looking at aspects of sequences and Linq in detail there are a few Linq methods we can use to simplify coding by ensuring that `IEnumerable` collections are properly presented as a sequence of *typed* entities - rather than just having the `type object`. The simple Linq methods we'll demonstrate, from the .NET `System.Linq` namespace will allow "first-pass" collection filtering and selection against which one can then easily pipe the resultant sequence into functions from the `List` or `Seq` (or other) modules. Again, for the moment, just think of `seq` as being a `lazy list` but bear in mind two points...

1. When we chain Linq members onto an `IEnumerable` collection we're essentially "operating" directly in the .NET Framework environment - not within the native F# environment. To a large extent this is why it makes

sense, to me, up to a point, to chain Linq filters, selects etc. directly against an `IEnumerable` - it allows the .NET "core" to execute the required filtering and selection directly before "passing" the filtered collection to F# for subsequent processing in the native F# environment.

2. Because .NET is presenting one with a "pre-filtered/selected" collection that F# interprets as a sequence then all of the elements of the sequence will have been evaluated by the time it "gets to F#". A sequence may be lazy but, if its elements have all been evaluated by .NET this doesn't get you any of the potential benefits of a lazy value type - which subject we consider in greater depth shortly.

Let's consider a practical example that "mimics" a query expression but does so by using `System.Linq` extensions to .NET objects - it is an approach I prefer as the syntax is more in line with F# and other .NET languages than that of a query expression. We have a requirement to filter the `GetCultures` (which is itself filtered by `SpecificCultures`) to only select, say cultures, where the culture name ends with `ZA` and thence "re-format" the output to produce what looks like a key-value pair where the key is the region name and the value is a `RegionsNames` instance. Thence, rather than using a comprehension, firstly, add the `open System.Linq` namespace to your script and execute it. Secondly add the following code...

```
let c =
  CultureInfo
    .GetCultures(CultureTypes.SpecificCultures)
    .Where(fun culture -> culture.Name.EndsWith("ZA"))
    .Select
      ( fun culture ->
          let ri = new RegionInfo(culture.LCID)
          ri.TwoLetterISORegionName,
          { English = ri.EnglishName
            Native = ri.NativeName }
      )
  |> List.ofSeq
```

I've indented this expression over multiple lines in a "non-standard" way because I believe this presentation makes it clear what's going on with "chaining" using the full-stop character. Consider...

- The class we're acting upon is `CultureInfo` - it's "separated" at the top of the chaining and piping. Notice that the forward pipe at the end is at the same level of indentation as the specified class, `CultureInfo`.
  - On the next line, indented a further two spaces, we encode the static method `GetCultures` using the `SpecificCultures` filter - as before.
  - On the next line we encode a Linq filter using the `Where` method. `Where` takes a lambda expression as its argument. The anonymous

function receives, as its argument - which I have named as the value type `culture`, each filtered `GetCultures` output item in turn and applies the action (anonymous function body) - being a filter that states that we're only interested in items where the `Name` property ends with the string `ZA`.

  - On the next line we include a Linq `Select` method. `Select` is **not** used to select rows - as in, for example, SQL, but rather to "re-shape" the input item; here the lambda expression function argument is an item of `type CultureInfo` that I expose by the value type named `culture`. Given an input of `type CultureInfo` I code a function body that outputs what essentially looks like a key-value pair - as we did in the list comprehension. I could suggest that you hover your mouse over the `Select` but what you see will horrify you! As I said, just think of `Select` as being used to re-shape an input type into an output type (these could be the same, I suppose, but apart from using it to emit a side effect I can't see any point in that!).

- From our perspective all of the above Linq code is executed as a "single statement" and the output, to F#, "looks like" a key-value pair - we'll consider this in detail shortly. What happens next is that the output of the chained `GetCultures`, `Where` and `Select` is piped into the function `ofSeq` in the `List` module. The function `List.ofSeq` takes a sequence as its argument and converts it to a list.

**Video**

30. Basic Linq Extensions.

Without concerning yourself about the signatures at this time just run the code to verify it gives us what we *expect* and have seen from our previous work. A useful aspect of this code in FSI and of the way we've entered the chaining, line by line, is that you can highlight from the declaration down across each statement in turn to just execute a "portion" of the whole expression so you can see the how the output varies through each link in the chain in the FSI output window.

## Coding a Simple Function

"While we're here" - I said that if you want to be confused you could try making out the Native names for the regions of India and China - but why modify the `ZA` string and then re-run the entire expression? Let's be lazy and code a simple function from our preceding Linq code example. Add the following to your script...

```
let selectNames (cultureCode : string) =
  CultureInfo
    .GetCultures(CultureTypes.SpecificCultures)
    .Where(fun culture ->
```

```
        culture.Name.Contains(cultureCode))
    .Select
      ( fun culture ->
          let ri = new RegionInfo(culture.LCID)
          ri.TwoLetterISORegionName,
          { English = ri.EnglishName
            Native = ri.NativeName } )
|> List.ofSeq
```

The function name is `selectNames` and it has a single argument named `cultureCode` which must be of `type string`. Intellisense will show you the type of function output is `(string*RegionNames) list`. This is equivalent to `list<string*RegionNames>` but, for signatures, F# always uses the former syntax. Note that, in a signature, one does not use a comma to delimit the key-value pair that we're generating since, in signature syntax, the comma is exclusively used as the "expected" delimiter in a comma separated list of arguments to a method. The function body is the same as the body of the earlier value type `c` except that...

1. Instead of using a `Name.EndsWith` we use a `Name.Contains`.
2. Rather than hard-coding the string `ZA` we just use the name of the argument value type `cultureCode`.

You can run this in FSI - you won't see much other than its signature though! We need to execute the function to see any "useful" output. Try these three...

```
let zaNames = selectNames "ZA"
let inNames = selectNames "IN"
let cnNames = selectNames "CN"
```

I'm not going to show you the output here - even if it weren't in Chinese - as there's quite a lot of it! There are 11 elements for South Africa, 19 for India (ignoring the output for Djibouti where some of its culture codes contain the string IN) and 5 for China.

## Tuples

I've been judicious in bypassing discussions about the output signature of our function and some earlier value types - the reason being that we now have to consider tuple types before we can understand the signatures.

In our latter list comprehensions and the Linq `Select`, we specified two, comma-separated values to be included within each element of the generated list. This is technically known as a **Tuple** - an ordered sequence of elements.

I've used the term ordered again in the definition, but remember, that *ordered* does not mean *sorted*.

In F# we just call any such a sequence a tuple. "Mathematically" we'd call such a sequence of...

- One element a *singleton*.
- Two elements a *couple*.
- Three elements a *triplet*.

**Video**

31. A basic function for displaying specific region names.

- and so on but, generally, an n-tuple. This is too onerous - we'll just stick to tuple regardless of how many elements are concerned. The term singleton we use but in a different context, later. Thus, we refer to any one of `1, 2` or `3, 2, 1` or `3, 5, 1, 4, 12, 2` as a tuple. In F#, we generally span tuples by the bracket pair `()`, although if you don't use the `()` encapsulation, F# will generally "put it in" for you - where it can, otherwise, the compiler will complain. So, we could have, for our earlier `yield`...

```
yield
  ( ri.TwoLetterISORegionName,
    { English = ri.EnglishName
      Native = ri.NativeName }
  )
```

This is a tuple of two elements; the first element is of `type string`, the second of `type RegionNames`. The fact that the elements have a different type, which we know is not possible for lists and sequences, gives rise to a very important and useful observation...

**Elements of a tuple can have any `type`.** Unlike lists, sequences and other F# collections where the `type` is applied at the *collection level* - so, for such, *every* element has to be of the **same `type`**.

The comma separated syntax of a tuple can lead to some confusion since, as we have seen in dealing with .NET classes we invariably have to instantiate a class or invoke a method by specifying a comma-separated list also spanned by `()` - but, that is not a tuple! To ameliorate this confusion between dealing with comma-separated arguments and tuples, F# uses, in its signature syntax, the asterisk `*` instead of a comma to designate a tuple. Knowing that...

- In your **Lists.fsx** script hover your mouse over the .NET Framework method that uses two replacement positional parameters, of `String.Format`. It has the signature: `String.Format(format:string, arg0:obj, arg1:obj) : string`.

- Hover your mouse over the declaration of our `selectNames` function in the script **RegionNames.fsx** - it's signature is: `val selectNames : cultureCode:string -> (string*RegionNames) list`.

To be explicit about the output type of our function we could thus code...

```
let selectNames (cultureCode : string) : (string *
...  RegionNames) list = ...
// or
let selectNames (cultureCode : string) : list<string *
...  RegionNames> = ...
```

You'll find I more often use the latter syntax in my code rather than the former.

In reviewing the signature for the dictionary it's now apparent that we are not dealing with a tuple since it reads as (hover the mouse over the declaration of `Regions`) `IDictionary<string, RegionNames>` whereas, if we use the same comprehension in creating a list the output signature becomes...

```
val : RegionList : (string * RegionNames) list
```

## Dealing with multiple Native Names

If we presume upon the basis of our requirement that what makes an element of a region list distinct is the Native name, then the implication is that excluding Native name from the `dict` key (as, previously we have just been using the region `Name` as our key) will result in us discarding a large number of "required" Native names. The `dict` has 142 elements, the region list has 535 elements (for lists the element count is yielded by the function `Length`) so we'd be losing up to 535 - 142 = 393 potentially distinct Native names.

We could try encoding the key of the dictionary as a tuple of region `Name` and Native name but rather than then using a `RegionNames` instance as our value let's just make the value the English name of the region - a string, since it's pointless to include the Native name in the record type as it will be included within the key. However, this begs the question of whether or not we should even be using a dictionary at this stage as, whilst, in theory, the key may be distinct, we'll have a lot of duplicates in values - being the English name and, furthermore, how do we make use of a compound key based upon `(string *string)`? As I'm not prepared to address, let alone attempt to answer these questions now (or, indeed, ever), let's instead start with a list of tuples - the tuple being `(regionName, englishName, nativeName)` so it has type `(string * string * string)`.

Let's re-work the code we had for the Linq `Select` for the value type `c` so it instead produces a list of this tuple triple. Add the following code to your script...

```
let regionTuples =
  CultureInfo
    .GetCultures(CultureTypes.SpecificCultures)
    .Select
      ( fun culture ->
          let ri = new RegionInfo(culture.LCID)
          ( ri.TwoLetterISORegionName,
            ri.EnglishName,
            ri.NativeName) )
  |> List.ofSeq
```

Notice the subtle differences? All I've done is remove the record type's element names and added a comma delimiter between the English and Native name properties to "convert" this into a tuple of three elements - I've even added the bracket pair `()` to highlight the presence of the tuple - although you can remove these since they are implied by F#.

## Sorting a List

The list output of `regionTuples` is still a bit of a "mess" - insofar as it's hard to visually inspect the list for potential tuple duplicates or "unexpected" tuple occurrences. We need to sort the tuples in the list so it becomes easier to visually inspect the content and there are three ways you can do this...

1. The quick and nasty way; we just use the `List` module function `sort` that takes as its only argument the list to be sorted. This type of sort operates against the entire concatenated list element - that is, in this case all three concatenated elements of the tuple in the order in which they are presented.

   Sometimes, as in our case of dealing with a compound element such as a tuple, we want to be more specific about which tuple elements we want to sort by. We can do this two ways...

2. We use the `List` module function `sortBy` that takes as its argument a lambda expression in which the anonymous function body specifies how an element is to be sorted.

3. If we're using Linq we can use the method `OrderBy` whose argument is the same lambda expression that you would use for `List.sortBy`.

Applying `List.sort` is trivial - you just add a forward pipe to it after the forward pipe to `List.ofSeq`. The lambda expression for a `sortBy` is not complex - it just takes some getting used to. The forward pipe to `sortBy` we would also add after the `List.ofSeq` as follows...

```
    |> List.sortBy (fun (code, english, native) -> code,
...   english )
```

When dealing with lambda expressions before we've only used a "shadow" value type of a single named argument - such as `elem`. In this case we know that `elem`, *per se*, is of type `(string * string * string)` - it's a tuple of three elements. Therefore, by specifying `(code, English, native)` we're instructing F# that we want the argument *decomposed* into its underlying tuple elements and to assign each element value, in turn to the named value type - so the first tuple element of `elem` will be assigned to `name`, the second to `english` and so on. Now, you'll see in the tuple decomposition in the script editor, that the shadow value type named `native` is in grey - as opposed to black; F# Power Tools has recognised that the named value type is subsequently unused. The question it poses it why bother declaring it? There is a place-holder that one can use in such an expression that tells F# that you don't care about the item - not going to use it, so just ignore it. That place-holder is the underscore character _ and, as you can likely guess, its meaning is slightly different in context to its usage as a catch-all in a pattern match case; in a pattern match it means *every other potential case* whereas, in an anonymous function argument, it means *ignore this positional argument*. We can thus re-code our `sortBy` as...

```
    |> List.sortBy (fun (code, english, _) -> code, english
...   )
```

Now, that's the function arguments used by the anonymous function of the lambda expression - what about the function body? Well, what you have to remember about functions is they take inputs and produce an output. It is the output that concerns us here - the output must be something on which F# can perform a sort - it must, implicitly or explicitly support, as Intellisense will tell you if you hover over the `sortBy`, the `Operator.compare` function. Here, we instruct F# to sort the output using a tuple comprising two of the input tuple's elements - `code`, then `english` (these are, effectively, concatenated in order to perform the sort). F# applies the `compare` function to each item in the element collection to determine the sort order based upon our `code,english` output requirement. We don't have to do anything else since F# "knows" how to compare strings to tell which one comes first alphabetically - as it does for numbers as well. Much later on we're going to have to provide our own `compare` function for types that F# doesn't implicitly know how to sort on.

Note that if we'd used `List.sort` then the sorting would occur across all three concatenated input tuple elements. If we were wanting to use Linq instead of `List.sortBy`, we'd have coded the following...

```
let regionTuples =
  CultureInfo
    .GetCultures(CultureTypes.SpecificCultures)
```

```
    .Select
      ( fun culture ->
          let ri = new RegionInfo(culture.LCID)
          ( ri.TwoLetterISORegionName,
            ri.EnglishName,
            ri.NativeName) )
    .OrderBy
      ( fun (code, english, _) -> code, english )
  |> List.ofSeq
```

We've simply chained a Linq `OrderBy` method - that uses the same lambda expression as the `List.sortBy`, onto the "original" expression.

**Video**

32. Tuples and sorting - Parts I & II

The foregoing should raise some questions in your mind about Linq because it looks as though it's tightly-coupled to F#. My take on it is that Linq isn't *actually* a part of the .NET Framework - it was "added" to it but developed separately. There's no doubt that it's tightly integrated into the .NET Framework but I have a suspicion that the F# code-base often-times seems to use Linq modules directly as opposed to going through the .NET Framework. It was, after all, developed following the implementation of Linq into .NET. To be honest, either way, I don't really give a damn - my interest is in producing effective, understandable code in a short time-frame and as long as performance is acceptable I don't care too much about what goes on in IL or elsewhere "under the covers".

Either way, sorting now enables one to visually browse the list in the likes of FsEye to get a feel for what one's dealing with and where potential problems may occur.

## A Warning about using the LCID

This issue about Djibouti is bugging me; I'd like to be fairly sure that I have data integrity before continuing onto actually using the output of our sorted region tuples as a "master" data set for subsequent usage. This needs to be investigated and resolved before I would be happy about having a "definitive" region list.

I've created another script file - **Djibouti.fsx** in the **fsx** folder. For the script include FsEye and the `System` and `System.Globalization` namespaces, then the following code...

```
let cultures1 =
  CultureInfo.GetCultures(CultureTypes.SpecificCultures)
  |> List.ofArray
  |> List.filter
```

```
        ( fun culture ->
6           culture.Name.ToUpperInvariant().EndsWith("ZA"))
    |> List.map
8       ( fun culture ->
            culture.LCID,
10          culture.Name,
            new RegionInfo(culture.LCID) )
```

For `cultures1` we're producing a list - so we can immediately see output in the FSI output window (as a list is not lazy) and then using a few pipes to yield what we want against the `CultureInfo` instances that are region-specific - those filtered by `SpecificCultures`. As usual we'll filter out cultures based upon where we know an error occurs - for South Africa with region code `ZA`. This time I'm using a `ToUpperInvariant` (from the `System` namespace - it is an *instance* member of the `String` class) to capitalise whatever the culture code is so I can be sure I'll catch `za` and variants as well as `ZA`. `ToUpperInvariant` capitalises its argument according to the English language specification (as per the definition of the invariant culture) whereas `ToUpper` would capitalise the argument according upon the rules of the underlying thread culture - which may not be English based.

I'm now using a `List.map` to re-shape my output (in the same way that a Linq `Select` would re-shape output) - I think the critical fields for inspection are the culture code, the culture LCID that is used to instantiate the `RegionInfo` class and the resulting `RegionInfo` instance for that LCID itself. If you run this in FSI you'll see the output...

```
val cultures1 : (int * string * RegionInfo) list =
2   [ (1078, "af-ZA", ZA); (7177, "en-ZA", ZA);
    (4096, "nr-ZA", ET); (1132, "nso-ZA", ZA);
4   (4096, "ss-ZA", ET); (1072, "st-ZA", ZA);
    (1074, "tn-ZA", ZA); (1073, "ts-ZA", ZA);
6   (1075, "ve-ZA", ZA); (1076, "xh-ZA", ZA);
    (1077, "zu-ZA", ZA)]
```

Clearly there's an issue with the culture code `nr-ZA`. `ZA` is South Africa, `nr` is for the Ndebele region/people - the isi prefix in the name just means the "*language of ...*". The tuple shows that the `RegionInfo` instantiated against the LCID of 4096 for the culture `nr-ZA` has a region code of `ET` - that is, Ethiopia - not quite Djibouti, but it is "next door" to Djibouti! I know the Ndebele are in South Africa - not Ethiopia nor Djibouti so, somewhere, there's been an error. Given the LCID of 4096 issue the following in your script...

```
let q1 = new CultureInfo(4096)
```

FSI will show you the output...

```
System.Globalization.CultureNotFoundException: Culture
...  is not supported.
Parameter name: culture
> 4096 (0x1000) is an invalid culture identifier.
```

Now, that's a surprise! *I* didn't originally specify `4096` - it's presumably buried in the Windows National Language Support (NLS) objects or in the .NET Framework. It's interesting in hexadecimal that it's `0x1000` so it looks like, to me, someone's placed an "upper limit" on the integer value for an LCID to be 1 less than that - that is, $2^{12}$-1, and that's still within the bounds of a "tiny integer". I'm not going to attempt to resolve such an issue because it seems to me this is a bug in the .NET Framework or in Windows 10 NLS. Thus, let's try a "workaround"; let's see what the following comes up with...

```
let cultures2 =
  CultureInfo.GetCultures(CultureTypes.SpecificCultures)
  |> List.ofArray
  |> List.filter
      ( fun culture ->
          culture.Name.ToUpperInvariant().EndsWith("ZA"))
  |> List.map
      ( fun culture ->
          culture.LCID,
          culture.Name,
          new RegionInfo(culture.Name) )
```

That is, we instantiate the `RegionInfo` based upon the culture name (code) rather than the culture LCID. The FSI output shows...

```
val cultures2 : (int * string * RegionInfo) list =
  [ (1078, "af-ZA", af-ZA); (7177, "en-ZA", en-ZA);
    (4096, "nr-ZA", nr-ZA); (1132, "nso-ZA", nso-ZA);
    (4096, "ss-ZA", ss-ZA); (1072, "st-ZA", st-ZA);
    (1074, "tn-ZA", tn-ZA); (1073, "ts-ZA", ts-ZA);
    (1075, "ve-ZA", ve-ZA); (1076, "xh-ZA", xh-ZA);
    (1077, "zu-ZA", zu-ZA)]
```

That's better; all the region codes now appear to belong to South Africa and we have no more than that expected via the cultures that refer to South Africa.

The moral of this story is two-fold...

1. If you get data integrity errors check them early on in the overall process.
2. Sometimes you'll have to find workarounds and sometimes that may mean a complete change in methodology rather than the type of simple change we just made.

Of course, if this is exposing a bug in the .NET Framework or the Windows 10 NLS and you have a later, updated version than me, then all of the above and prior observations about Djibouti will be meaningless to you as you wouldn't even have seen this behaviour! Even so, bear the prior two points in mind.

From our viewpoint we will not, henceforth use the culture LCID but rather use the culture code in dealing with the `System.Globalization` namespace.

## Category Totalling - Classic & Contemporary

Many, many years ago when developing with IBM Application System in the 80's there was what proved to be a very useful technique called **Category Totalling** which could be used in inline code and even Transact-SQL when a DB2 or SQL/DS Group By wasn't "quite sufficient", especially when producing data sets for reports and ad-hoc queries - which "technology" was apparently only invented by "marketing consultants" and branded as "data warehousing" in the 90's! I don't know if there's a "modern" equivalent or what some "marketing consultant" may call it nowadays so I'll continue to use the terminology I'm familiar with.

The principle is as follows...

1. You sort the source data set in the key order against which you wish to perform sub-totalling or collation of data like a Transact-SQL group by. In our case this will be the region code and English name. As we've just seen, we can use a sorted list to give us a "set" of Native names where the region code and English names are the same.

2. You assign a value type against the key or keys you wish to detect a change for in the streamed, sorted source data and you give, as its initial value, the value(s) present in the first row of your source data. Clearly, the value type has to be mutable since we expect, in any "decent" data set, the key value to change so we have to be able to "update" this value type.

   The value type usually has a name like `lastKey`. If there's more than one key one wishes to subtotal data by, then we'd just use the prefix `last` and then the "column" name. Here that is the region code and English name from the "incoming" tuple and their initial value will be that in the `Head` of the input list ot tuples.

3. We don't need to "detect" changes in Native name but we *do* wish to perform an operation on every row of the input source where the key(s) is(are) the same - we wish to concatenate all the Native names for matching key(s). We therefore define an "accumulator" variable for this purpose. Generally such a value type is named as `acc`.

4. You then "read" through each sorted source row sequentially and compare the `last` key(s) with the *current* key(s) for the incoming row. For the first row these will clearly be the same since that's how we initialised our `last` key(s). For each incoming sorted row...

    i. In the event that a `last` key has the same value as the equivalent *current* key...

        - You're still dealing with a row that has the same key value so now you would invoke whatever process you need to in order to "total" or concatenate (accumulate) the current row with the previous one.
        - For our purposes we would define a, clearly mutable, value type that would contain the concatenated names, our "accumulator" and carry this value across each subsequent row that has the same key when invoking some function to perform the actual concatenation (accumulation).
        - Once you've invoked what function you need to for the current row (which has the same key value as the previous row) you're done with the current row so you iterate to the next row in the source data set.

    ii. If a `last` key and equivalent *current* key have different values...

        - You have detected a change of key - in our case the current row has a different region code and/or English name to that of the previous row.
        - The first thing you do is write out your `last` key and any "carried" value types that hold accumulations for the previous rows with that `last` key into your output data set.
        - This is the primary reason why you **must** sort your input dataset into key order; if you've already written out your accumulation value types and subsequently come across another input row that has the "same key(s)" as one you've finished processing with beforehand, you can't access and modify your previously evaluated accumulator values for that key.
        - Having written your output row...
            *a.* You must then set the value of the `last` key to be the value of the key of the *current* row.
            *b.* You should re-initialise your accumulator value types. For us this will mean re-assigning it on the basis of the the current row region code and English name.

5. When you reach the end of the source data set you'll have a "hanging" last key and your accumulator value type(s) - you can't detect if there is a *subsequent* key change because you've got no more input rows to process, so you just write these values directly into your output data set.

This is the "classic" methodology and, as you can see, it's fairly involved. Furthermore, we're explicitly introducing the concept of the mutable value type and, associated with that, something in F# known as a reference value type. These use the F# keywords `mutable` and `ref`. Prior to Version 4.0 of F# we'd have to take account of the subtle differences of mutability and referencing and make sure our code conformed in an "appropriate manner" to the guidelines about how to and, more importantly, when to use one or the other. Fortunately, in Version 4.0 of F# - we no longer care! We just use the keyword `mutable` and the F# will compiler will make the choice, automatically and "invisibly" as to whether it should re-define a `mutable` type to a `ref` type in the compiled IL.

## Mutable Value Types

In dealing with this classical methodology the code may make you cringe; there's a lot going on but, because the code is very much "sequential" one can inspect staged output in FSI/FsEye and, given the nature of the data we're dealing with, it should be relatively simple to scan through intermediate lists to see what's going on. Furthermore, it's an educational exercise!

Given the requirement for a category totalling operation it's apparent that we can't use an immutable value type to store the result of our "accumulator function" or "last key" value(s) and thereby respectively modify the result by concatenating additional Native name strings for a subsequent list element with the same region code or change what is "referenced" as the last key value we dealt with - that is, for the previous list element(s).

Let's firstly consider this process whereby we can concatenate names into a text string across list elements that have the same region code: We'd like to concatenate all Native names together into a string where each name is delimited by ";". We actually have to undertake this process a number of times; once in initialising our accumulator value type for the region code/English name category totalling keys and then, when dealing with each list item coming in, dependent upon whether the "current" items' keys are the same as those of the previous list element. This is the point mentioned previously when the last key(s) and the current key(s) are either the same or different. Because we're invoking this process three times from different "locations" we'll code a function that can merge Native names into an "accumulator".

We'll put this in a new script file called **Folding.fsx** in the **fsx** folder. At the top of the script ensure you open the namespaces `System` and `System.Linq` - just in case we'll need them and, if you wish, also include the FsEye snippet. For my part I also reset the FSI session and then execute the FsEye snippet and the `open` statements.

For a function that concatenates Native names let's code the following...

```
let mergeNames (accumulated : string) english native =
  if not <| String.IsNullOrWhiteSpace(native) then
    if native <> english &&
       not <| accumulated.Contains(native) then
      if not <| String.IsNullOrWhiteSpace(accumulated)
        then String.Concat(accumulated, ";", native)
        else native
    else accumulated
  else accumulated
```

**Video**

33. Category Totalling accumulation function.

For this code we have a function named `mergeNames` that takes three curried arguments and we've had to help the F# compiler by explicitly specifying that the curried argument named `accumulated` is a string; if you omit this explicit typing F# will complain that it doesn't know what `accumulated` is so it can't be sure it can apply the method `Contains` against it. Thence...

- We check that the parsed Native name in `native` is neither a `null` nor an empty string nor consists only of whitespace. In the future I'll just refer to this as "is a valid string".
  - If `native` is not a valid string then control passes to the `else` expression which just issues, as the function output, the untouched value of the `accumulated` argument.
  - If `native` is a valid string...
    - It's possible that the Native and English names are the same (for example as for United Kingdom with culture code `en-GB`) - we want to ignore such Native names (no point in duplicating names).
    - It's possible that our `accumulated` string already contains the Native name.
      - If either of the above are `true` then we just output the accumulated string as is.
      - Otherwise we validate that `accumulated` is a valid string: If so, concatenate, using `System.String.Concat`, the Native name to the `accumulated` string delimited by a semi-colon. If not, we just output the Native name since we don't want the first Native name in the accumulated string to be prefixed by a comma.

We can briefly test this function by first executing the function declaration then via...

```
mergeNames String.Empty "South Africa" "Suid-Africa"
mergeNames "South Africa,Suid-Africa" "South Africa"
...  "Aforika Borwa"
```

Here, to be explicit, rather than coding `""` as our initial accumulation value I've used the static member `String.Empty` to represent an empty string. This code thus copes with an "invalid", initial accumulation string and does tack Native names onto the end of the `accumulated` string.

**Video**

34. Category Totalling - source data.

Let's now assign and initialise our "last key" value and accumulator value types. For this we'll use the `regionTuples` code from our **RegionNames.fsx** script and paste it into **Folding.fsx**. You'll also need to put an `open System.Globalization` into your script and execute that before "playing" with `regionTuples`. This gives us our sorted list based upon the composite key of the list tuples' first two elements of region code and English name. Enter the following after you've copied, pasted and executed `regionTuples`...

```
let (firstCode, firstEnglishName, firstNativeName) =
...  regionTuples.Head
let mutable lastKey = (firstCode, firstEnglishName)
let mutable acc = String.Empty
```

For the above...

- I'm de-composing the `Head` of the `regionTuples` list into three distinct named value types so that I can subsequently refer to the elements by name directly. The head of `regionTuples` is the tuple `("001", "World", "World")` - so the value `001` is mapped to the value type named `firstCode`, `World` is mapped to the value type named `firstEnglishName` and `World` is mapped to the value type named `firstNativeName`.
- I declare a `mutable` value type named `lastKey` whose initial value is the tuple of the "input" lists' first region code and English Name.
- I declare a `mutable` value type named `acc` whose initial value is just an empty string.

You can run these three expression in FSI to view these initial values.

**Video**

35. Mutable value types.

Now, without "giving the game away" what we'll use is the `List` modules' `collect` function. What `collect` does, according to the MSDN documentation, is "*applies the given function to each element of the sequence and concatenates all the results*". This is the usual cryptic "note" that you'll find in the documentation and, if, at this stage, you analyse the function signature to try and figure out what it does you may find that even more confusing. It takes a while to get used to signatures. Let's try to explain

this in more understandable terms: Firstly, `collect` takes a lambda expression as its first argument - as do pretty much all collection related functions. Secondly, `collect` does not use a built-in accumulator (we'll see shortly that the likes of the `fold` function do). The function takes each element of an input sequence, the second argument of `collect` (which is usually provided by a forward or backward pipe) and applies the code in the lambda expressions' anonymous function body to it. Well, such is no different from other collection functions like `iter` that is used to iterate through a collection. The difference with `collect` is that for each input element you must specify an output element *of the same type*. Since our input is a list of tuples we must provide, for each element we process, an output list of tuples. Since the `type` is tuple, how many or few elements go into the input/output is up to you - so you can greatly "expand" the incoming list or, as we wish to do, reduce it in terms of its tuple element count. Once `collect` has finished processing all of the input elements it will then join all of the output lists that you produce - one for each incoming element, into a single list - so the `collect` output, like the input, is a *single* list rather than a list of lists!

**Video**

36. List.collect - how we specify function output.

To see this "in action" what we're going to do is provide another mutable value type and that will contain the output list of tuples to produce from the anonymous functions' consideration of the input element. The essential difference between the input list of tuples and the output list of tuples is that the latter won't contain a single Native name - it will contain our `accumulated` string of Native names. Some of you may hear alarm bells ringing; surely that will produce a "steadily growing" accumulation of Native names for each key! Indeed, it would - unless one is judicious in determining *what* to output and, importantly, *when* to output it and also remembers the existence of the empty list `[]`!

Firstly, add the following declaration to your script...

```
let mutable outputTuple = regionTuples.Head
```

The initial value for `outputTuple` is irrelevant - we specify it only because it tells the F# compiler "what to expect" - the *actual* values we'll be assigning in the lambda expressions' function body - as for our accumulator.

Now, the meat of the code; add the following value type declaration to your script...

```
let Regions =
  regionTuples
  |> List.collect
      ( fun (code, english, native) ->
          let currentKey = (code, english)
          if lastKey <> currentKey then
```

```
                outputTuple <- (fst lastKey, snd lastKey, acc)
                lastKey <- (code, english)
                acc <- mergeNames String.Empty english native
                [outputTuple]
            else
                acc <- mergeNames acc english native
                []
    )
```

> **Video**
>
> 37. Category Totalling - how old-style works.

It's perhaps trite to say "that's all there is to it" because you have to follow this really closely until you get to grips with the `collect` function. Firstly, we take our list of tuples, `regionTuples` and forward pipe that into the `List.collect` function; `collect` requires two arguments - the first we give it is the lambda expression, the second is the piped `regionTuples` list. The anonymous function receives, as its argument, each input list element in turn which, as we know it's a tuple, we'll immediately decompose into some named value types which are re-assigned on each "pass" through the input lists' elements.

The function body does the following...

- We assign a value type named `currentKey` that is a tuple comprised of the current processed elements' region code and English name; no need for mutability here - this is re-assigned for each input element.
- We then compare the `lastKey` value with the `currentKey` value. Since we pre-declared `lastKey` to have a value assigned from the `Head` of the `regionTuples` then this ensures that this condition will never be met for the first element in the input list.
    - If the `lastKey` and the `currentKey` do not match, then we have detected a change in key for which we must perform a category total...
        - Update our mutable `outputTuple` to contain the values for the last processed key - that is, not the key of the *current* element (which is different from that of the preceding element) but the key of the preceding element(s) for which we have accumulated the Native names. Because `outputTuple` is mutable we don't use the equals sign we use the operator `<-`.

          The other thing you'll see in the code is the method of assignment of the first and second elements to the `outputTuple` value; `fst` and `snd` are actually F# functions that take a single argument - a tuple and output, respectively, the *first* and *second* element of the input tuple. That's all you get - there is no third, fourth etc.!
        - Change the value of `lastKey` to be the `currentKey`.

- Invoke `mergeNames` but with an initial empty string to re-start the accumulation of Native names using the current keyed Native name.
- Output, for `collect`, a list that contains the *single* tuple maintained in `outputTuple` - so this will *always* be a list with a *single* element whose tuple consists of the evaluated category total for the preceding key value. To "convert" `outputTuple` to a list of tuples we just encapsulate it with `[]`.

- If the `lastKey` and the `currentKey` match, there is no change in key value so we...
  - Update the accumulator through the mutable value type with the current keys' Native name via the `mergeNames` function using the existing accumulator string value.
  - We output, for `collect`, an empty list. This is very important - we *have* to output a list so we may as well make it the empty list since we're not interested in this current element being included into our ultimately "compressed" output list for `collect`.

**Video**

38. Running old-style category totalling.

You can run this in FSI but we haven't quite finished: What about the *last* `lastKey`? Well, what is the last tuple in `regionTuples`? You can find this out two ways...

1. You can use FsEye to inspect the `GetEnumerator` node; you'll have to expand this node several times because FsEye only returns 100 sequence elements at a time and we need to look at the 533rd element!

2. You can type the following into the FSI output window `regionTuples|> List.rev |> List.head;;` and press **Enter**. We take the list, pipe it into the `reverse` function so the last element becomes the first and then take the `head` of the reversed list.

**Video**

39. Tacking on the last accumulated element of the input list.

Either way, the last tuple is `("ZW", "Zimbabwe", "Zimbabwe")`. Now, in FsEye, if you look at `Regions`, try as you might, you'll not see this tuple in the list. We have to write this `lastKey` based output tuple into the `collect` output list ourselves. We can do this using the function `List.append` - you can try this by adding the following forward pipe to the declaration of `Regions` after the collect...

```
|> List.append [(fst lastKey, snd lastKey, acc)]
```

Effectively we're just manually replicating the assignment of the `outputTuple` in the earlier listing and "wrapping" the tuple in a list. If you run the modified `Regions` code now, you'll see that the Zimbabwe tuple appears at the head of

the `Regions` list. If you really find that offensive and want it to appear "where it should", as the last element of Regions, you could do the following pipe sequence instead...

```
|> List.rev
|> List.append [(fst lastKey, snd lastKey, acc)]
|> List.rev
```

This, however, doesn't strike me as an "efficient" exercise so I would advise the following instead...

```
let Regions =
  List.append
    ( regionTuples
      |> List.collect
          ( fun (code, english, native) ->
              let currentKey = (code, english)
              if lastKey <> currentKey then
                outputTuple <-
                  (fst lastKey, snd lastKey, acc)
                lastKey <-
                  (code, english)
                acc <-
                  mergeNames String.Empty english native
                [outputTuple]
              else
                acc <- mergeNames acc english native
                []
          )
    )
    [(fst lastKey, snd lastKey, acc)]
```

In the latter block of code, we specify `List.append` *first*, then `List.collect` output against `regionTuples` as its `first` argument and then tack on, as the second argument, the list comprising the last `lastKey`. No reversal or additional processing is thus required.

**Video**

40. A more efficient List.append.

## Doing it "New Style"

"Old style" category totalling probably died on .NET capable platforms when Linq came out as part of the .NET Framework - this was in November 2007 as part of the .NET Framework Version 3.5. Who knows what they're now using instead on IBM/Application System (that was in the mid to late '80's and they're probably still using it)! I found with the introduction of Linq in .NET that, especially using C#, it was really complex to do the likes of category totalling

because one had to delve deeply into "generics" and how some of these "more complex" functions could even be constructed from within C#. Now, in F#, this kind of Linq functionality is pretty simple regardless of whether you do it with native F# or `System.Linq` extensions since, as we've seen, the lambda expression would be the same. That said, in looking at our Type Providers in Volume II, I do make use of "classic" category totalling to re-shape a collection of data.

Some say that one of the "hardest" functions in F# to grasp, when you're learning, is the `fold` function as it applies to F# collections - lists sequences etc. I tend to disagree; if you can follow something like `collect` then `fold` will hold no problems and I really think that the "issue" is the lack of holistic examples that use the likes of `collect`, `fold` etc. Now, to show you "new style", we're going to use three more F# collection "common" functions - `fold`, as you might have surmised, `map` and `groupBy`. Unlike previously, I'm just going to "dump" the code so you can inspect it before we go through it. Add the following to your script...

```
let distinctRegionsList0 =
  regionTuples
  |> List.groupBy
      ( fun (code, english, _) ->
          (code, english)
      )
  |> List.map
      ( fun ((code, english), childList) ->
          code,
          english,
          List.fold
           ( fun (acc : string) (code, english, native) ->
               if not <| acc.Contains(native) &&
                  native <> english
               then acc + native + ";"
               else acc
           )
           String.Empty
           childList
      )
```

This is all there is! Let's go through the `groupBy` first; because of the way I've structured the statement you can just highlight the declaration and the groupBy expression and execute that in FSI "stand-alone". If you do that your output for Spain with region code `ES` will resemble...

```
  ( ("ES", "Spain"),
    [ ("ES", "Spain", "Espanya"); ("ES", "Spain",
...  "Espanya");
```

```
        ("ES", "Spain", "España"); ("ES", "Spain",
...  "Espainia");
        ("ES", "Spain", "España")
    ]
);
```

I find this very instructive; `groupBy` has taken our "sorting" tuple of the region code and English name and put that as the first element of *another* tuple whose second element is actually a list of all of the elements in the input list which have the same region code and English name "key" (the first element). The entire element, as displayed in the foregoing FSI output, is a single element of a `groupBy` output list which is "keyed" by the key combination of region code and English name.

From the point of view of the code you can see that the lambda expression anonymous function takes as its argument the decomposed tuple of the input list element and from that, the function body produces a tuple that is the `groupBy` "key" for the output list. We haven't told it, anywhere, to produce a list of all the input lists' elements that have this "grouping" key - it just "knows" to do this.

**Video**

41. Category Totalling - new style. `List.groupBy` and `List.map`.

Now that we have a distinct key to "group" the input data by, the second element, being a list of all the original elements that share the same grouping key, is amenable to a collection function called a `fold`. In order to do the `fold` we first have to tell F# that we want to perform some processing against every element of the input list - which is what is piped to `map` as the output of `groupBy`. `map` is similar to the Linq `Select` we looked at previously - it allows you to "re-shape" the list elements, even change the element (and therefore the list) type but doesn't permit you to reduce or enlarge the number of elements in the list.

Let's consider the argument list of the lambda expression anonymous function for `map`; you can see that it represents the structure of the `groupBy` list output element - a tuple whose first element is itself a tuple being the region code, English name "key" tuple, and whose second element is a list that I've named as `childList`.

For the `map` my function body dictates that I'll be re-shaping the input to produce an output that will be a tuple of three elements...

1. The region code,
2. The English Name,
3. The output of the `List.fold` function.

**Video**

42. Category Totalling - new style. Running the code.

A `childList` element is a complete `(code, English, native)` tuple whose code and English values match those of the "key" tuple. What we'd like to do is fold the `childList` to accumulate the Native names and just return a single string that is the concatenated Native name list for all of the list elements that share the grouping key. This is what `fold` does; as a function it takes three arguments...

1. A lambda expression that specifies how accumulation is to take place to yield an accumulation value type. The anonymous function takes two arguments - which makes it different from the other collection functions we've seen to date...
    i. The first argument is the accumulator which value I've mapped to a value type named `acc`.
    ii. The second argument is the structure of the input element of the input list which is to be accumulated in some fashion. Here I know the input element (which is a `groupBy` keyed list) is the "full" region tuple of `(code, English, native)`.

    The anonymous function body is the block of code that determines how the accumulator should function. In this case I want `acc` to be a string of concatenated Native names so I specify that...
    i. If the native name is not the same as the English name in the input element tuple and the Native name does not yet appear in the accumulated value, so add the Native name to the accumulator and stick a semi-colon on the end of it.
    ii. If the foregoing is not true, then we don't want/need to change the accumulator so just output the `acc` value.
2. The second argument `List.fold` requires is the initial value for the accumulator `acc` - we just give it an empty string to "kick-off" with.
3. The third and final argument that `List.fold` requires is the list to fold against! You'll "usually" see that this argument is piped into `fold` rather than specified "in full" the way we've done so here.

If you run this in FSI you'll see that the output is similar to that of `Regions` in category totalling "old style", except that we have a trailing semi-colon for our concatenated Native names. Let's get rid of this - try the code...

```
let DistinctRegions =
  regionTuples
  |> List.groupBy
      ( fun (code, english, _) ->
          (code, english)
      )
```

```
    |> List.map
8       ( fun ((code, english), childList) ->
          code,
10        english,
          ( List.fold
12            ( fun (acc : string)
                    (_, english, native) ->
14                if not <| acc.Contains(native) &&
                     native <> english
16                then acc + native + ";"
                  else acc
18            )
              String.Empty
20            childList
          ).TrimEnd(';')
22      )
```

I know `List.fold` produces a string so I've just put brackets around `List.fold` so F# executes that then, for the output, I use the `TrimEnd` method with an argument of a semi-colon. Note that `TrimEnd` uses `char` as the argument type not string - hence I have to specify the single character `';'` rather than the string `";"`.

**Video**

43. Trimming the native name concatenation.

Just to be sure, type `Regions = DistinctRegions;;` in the FSI output window and press **Enter**. The response will tell you if the "old style" output is identical to the "new style" output and the answer is **true** - yes, they are the same. On a final "new style" spin - let's try this as a sequence - even though we haven't "covered" `seq` and let's ignore the sort that was done of the tuple list region tuples - `regionTuples`. We can code the whole category totalling process in the single value type assignment as follows...

```
let DistinctRegionsSeq =
2   CultureInfo.GetCultures
      (CultureTypes.SpecificCultures)
4   |> Seq.map
      ( fun culture ->
6       let ri = new RegionInfo(culture.LCID)
        ri.TwoLetterISORegionName, ri.EnglishName,
...   ri.NativeName
8     )
  |> Seq.groupBy
10    ( fun (code, english, _) ->
        (code, english)
12    )
```