```
 1 # Gherkin Translator
 2
 3 The translator converts a Gherkin
 4 feature file into a unit test file.
 5 The unit test file calls a glue file which
 6 the developer edits to call the code under

 7 test.
 8 The translator also creates a template for
    the
 9 glue file.
10
11 The translator is a single file containing
    all
12 the necessary components.  To translate a
13 feature file, run it from the IDE.
14
15 Unlike other implementations of Gherkin,
16 each feature file is associated with one
   unit
17 test file and its glue file.
18
19 ## Why Use Gherkin?
20
21 Gherkin feature files are readable
   executable documentation.  If a
   requirement / story includes buiness rules
   ,
22 they provide a convenient form to
   collaborate with non-programmers.  For a
   developer they provide an alternative way
   to specify the values used to call a
   parametrized test method.
```

```
23
24
25 ## Why Not Use Existing Frameworks?
26
27 I've been using Cucumber, one of the most
   common applications that uses
28 Gherkin for a number of years.  You can
   have a table after each step.   However
   you
29 need to add additional code to use that
   table as a list of objects.  The means for
    doing
30 so has changed from version to version.
   The code for doing so has gotten more
   complex.
31
32 ## Example
33 Let's give an example of how this works.
34
35 Here is a feature file.  In the test
   directory, it is named "examples.feature".
     The words after the keyword
36 `Feature` are combined into the name of
   the feature.  Let's assume that you are
   using the translator with Kotlin (language
    suffix .kt)
37 The operation is the same, the output code
    depends on the language.
38
39 The single step in the `Scenario` ("
   Convert F to C ") is passed a list of
   objects of
40 `TemperatureComparison`.
```

```
41 A unit test file with the name `
   Feature_Examples.kt` (with language
   appropriate suffix) is created in a
   directory with the same name.
42 A another file is created called `
   Feature_Examples_glue.impl` is also
   created.  This contains code that is
   called
43 from `Feature_Examples.kt`.
44
45 The first time you run the Translator, you
    should rename that file to the language
   appropriate suffix
46 (e.g. rename it from .tmpl to .kt).  You
   will be making changes in this file to
47 call your production code.  If you add new
    steps to the feature, you can copy a
   template for the new steps from
48 the template file (.impl) to the glue
   source file (.kt).  Alternatively, you can
    just let the IDE suggest that you need
49 a new method in  Feature_Examples_glue.
50
51 The two words after the comment sign #
   denote that data format that is passed to
   the glue code and the class name.
52 For `Feature_Examples`, the table values
   will be converted to a List of objects of
   type `TemperatureComparison`.
53
54 ```
55 Feature: Examples
56
```

````
57 Scenario: Temperature
58 # Business rule , Calculation
59 * Convert F to C # ListOfObject
   TemperatureComparison
60 | F    | C    | Notes       |
61 | 32   | 0    | Freezing    |
62 | 212  | 100  | Boiling     |
63 | -40  | -40  | Below zero  |
64 ```
65 The `*` is a keyword in Gherkin.  In the
   translator, you could also use `
   Calculation` or `Rule` instead.
66 This turns into code in  `Feature_Examples
   .kt`:
67 ```
68 class Feature_Examples{
69     @Test
70     fun test_Scenario_Temperature(){
71         val feature_Examples_glue_object
    = Feature_Examples_glue()
72
73         val objectList1 = listOf<
   TemperatureComparison>(
74             TemperatureComparison(
75                 f = "32",
76                 c = "0",
77                 notes = "Freezing",
78                 ),
79             TemperatureComparison(
80                 f = "212",
81                 c = "100",
82                 notes = "Boiling",
83                 ),
````

```
 84              TemperatureComparison(
 85                  f = "-40",
 86                  c = "-40",
 87                  notes = "Below zero",
 88                  ),
 89              )
 90          feature_Examples_glue_object.
    Star_Convert_F_to_C(objectList1)
 91          }
 92
 93 ```
 94 Now to simplify creation of the objects
    in a table, you create a data description
    .
 95 The `Data TemperatureComparison` portion
    produces code in the test file that
 96 declares a `TemperatureComparison` class.
      Every attribute in this class is `
    String` type. Since the table also
    contains the data types for each element
 97 in this class, a second class with the
    default name `
    TemperatureConversionInternal` is also
    created.  You can attempt to
 98 create an instance of this class in the
    glue code to check that the format of
    each element in the table is acceptable.b


 99
100 ```
101 Data TemperatureComparison
102 | Name   | Default  | DataType  | Notes
      |
```

```
103 | F       | 0          | Int
            |          |
104 | C       | 0          | Int
            |          |
105 | Notes   |            | String
        |          |
106 ```
107 turns into code in `Feature_Examples_data
    .tmpl` that looks like
108
109 ```
110 data class TemperatureComparison(
111     val f: String = "0",
112     val c: String = "0",
113     val notes: String = "", ) {
114     fun toTemperatureComparisonInternal
    () : TemperatureComparisonInternal{
115         return
    TemperatureComparisonInternal(
116             f.toInt(),
117             c.toInt(),
118             notes,)
119     }
120
121 data class TemperatureComparisonInternal(
122     val f: Int= "0".toInt(),
123     val c: Int= "0".toInt(),
124     val notes: String= "",)
125 ```
126 The first time you run the Translator,
    rename this file
127 The other file that is created is `
    Feature_Exmaples_glue.tmpl"
```

````
Again, just rename this file to language suffix.
```
class Feature_Examples_glue {

    fun Star_Convert_F_to_C( value : List<TemperatureComparison>) {
        println("---  " + "Star_Convert_F_to_C")
        println(value)
        fail("Must implement")
    }

```
Now comes your part.  Add the appropriate code to this glue function
to call the code you create.  If you only have one row,
then you might just code that one. The string values are converted into the
internal values.  The `F` value is passed to the `TemperatureCalculations.convertFarenheitToCelsius()`
method and the return value is compared to the `C` value.

```
    fun Star_Convert_F_to_C(value: List<TemperatureComparison>) {
        element = value[0]
        val temp = element.toTemperatureComparisonInternal()
        assertEquals(
````
````

```
150                 temp.c,
151                 TemperatureCalculations.
    convertFarenheitToCelsius(temp.f),
152                 temp.notes
153             )
154         }
155     }
156
157 ```
158 The compiler would suggest you create a
    method such as:
159 ```
160 class TemperatureCalculations {
161     companion object {
162         fun convertFarenheitToCelsius(
    input: Int): Int {
163             return ((input - 32) * 5) / 9
164         }
165     }
166 }
167 ```
168 Now you could change it to use every row
    in the table:
169 ```
170     fun Star_Convert_F_to_C(value: List<
    TemperatureComparison>) {
171         for (element in value) {
172             val temp = element.
    toTemperatureComparisonInternal()
173             assertEquals(
174                 temp.c,
175                 TemperatureCalculations.
    convertFarenheitToCelsius(temp.f),
```

```
                    temp.notes
                )
            }
        }
```
Note you can have as many columns and rows in the table as you need.
The form in the glue code looks the same - iterate around each row.

Inspiration