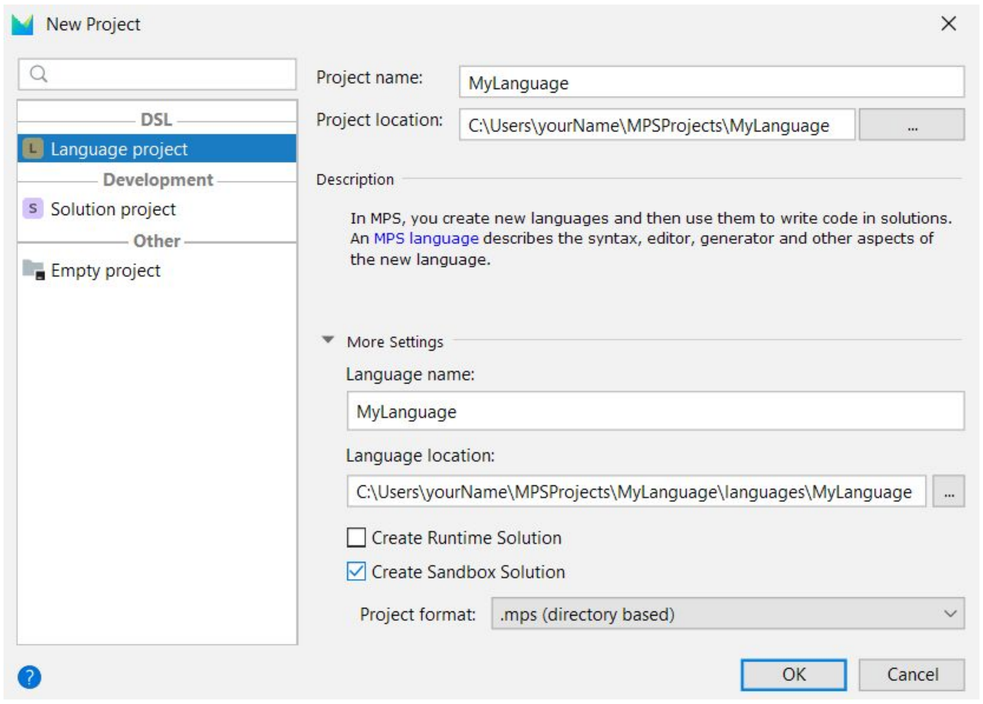
Creating C# Based MPS DSL

In this tutorial, you will create a very simple DSL based on the C# base language.

Please, follow the same initial steps as in the tutorial presented in Section *4.1* until you get to the *New Project* window and, according to that tutorial, you should create a solution project. This is the point from where this tutorial goes its own way.

In *New Project* window, you should select *Language project* and type in your project's and language's name, as in Figure 4.12. You may also want to check the *Create Sandbox Solution* checkbox if you want to experiment with your language while designing it. This is really recommended. When done, push the *OK* button.



New Project

Project name: MyLanguage

Project location: C:\Users\yourName\MPSProjects\MyLanguage

Description

In MPS, you create new languages and then use them to write code in solutions. An MPS language describes the syntax, editor, generator and other aspects of the new language.

More Settings

Language name: MyLanguage

Language location: C:\Users\yourName\MPSProjects\MyLanguage\languages\MyLanguage

☐ Create Runtime Solution

☑ Create Sandbox Solution

Project format: .mps (directory based)

Figure 4.12: Creating a language project

Go to the project explorer and expand the item *MyLanguage* with the yellow icon with the capital letter *L*. Then, right-click the *Structure* item, select *New* and *Concept*. As illustrated by Figure 4.13. This will create a concept, i.e. the modelling description for AST nodes that users of your language will create.

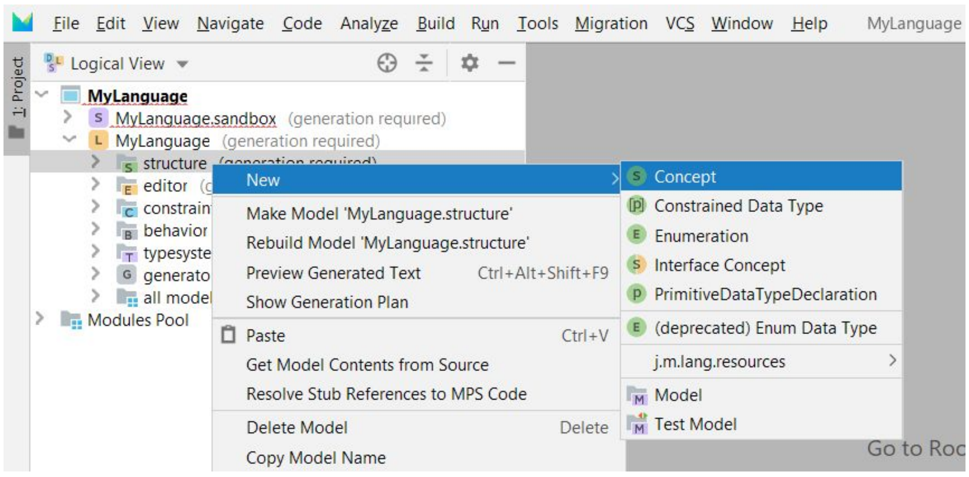
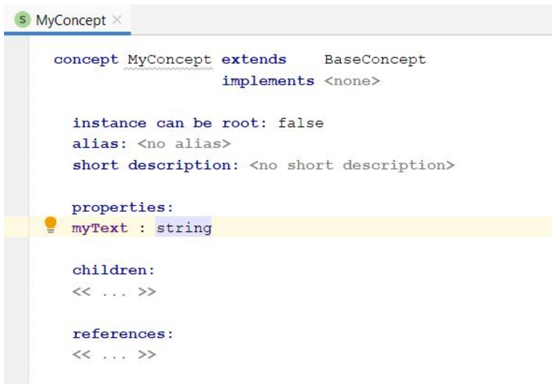

File Edit View Navigate Code Analyze Build Run Tools Migration VCS Window Help MyLanguage
1: Project
Logical View
MyLanguage
MyLanguage.sandbox (generation required)
MyLanguage (generation required)
structure
editor
constrain
behavior
typesyste
generato
all model
Modules Pool
New
Make Model 'MyLanguage.structure'
Rebuild Model 'MyLanguage.structure'
Preview Generated Text Ctrl+Alt+Shift+F9
Show Generation Plan
Paste Ctrl+V
Get Model Contents from Source
Resolve Stub References to MPS Code
Delete Model Delete
Copy Model Name
Concept
Constrained Data Type
Enumeration
Interface Concept
PrimitiveDataTypeDeclaration
(deprecated) Enum Data Type
j.m.lang.resources
Model
Test Model
Go to Roc

Figure 4.13: Creating a language concept

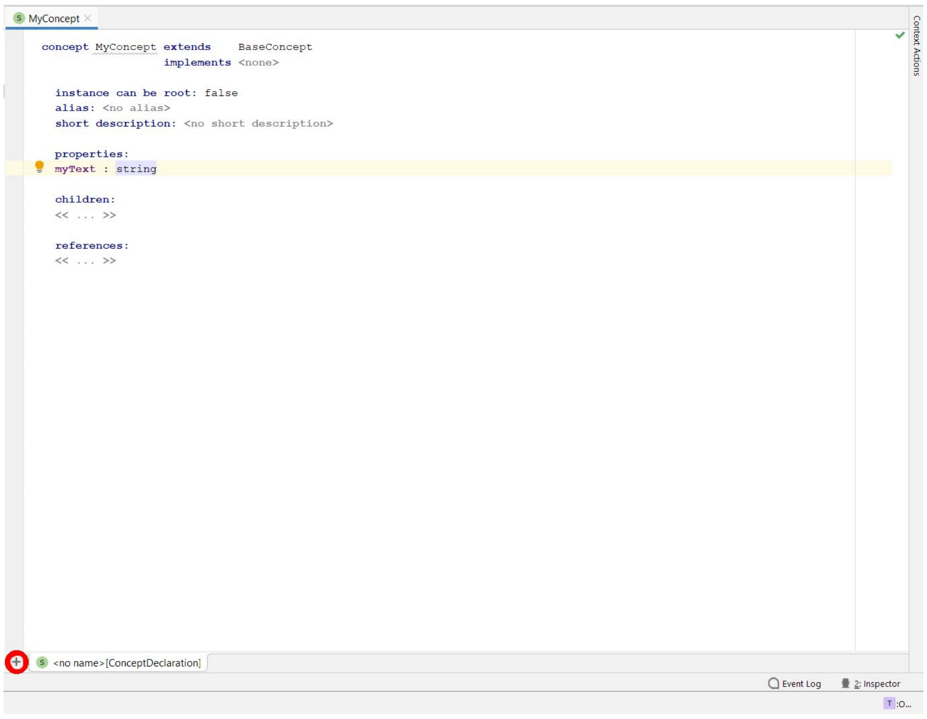
Now, fill in the name of the concept and, for example, add a string property for it. You can inspire by Figure 4.14.



```
concept MyConcept extends    BaseConcept
                  implements <none>

  instance can be root: false
  alias: <no alias>
  short description: <no short description>

  properties:
  myText : string

  children:
  << ... >>

  references:
  << ... >>
```

Figure 4.14: Initializing the created concept

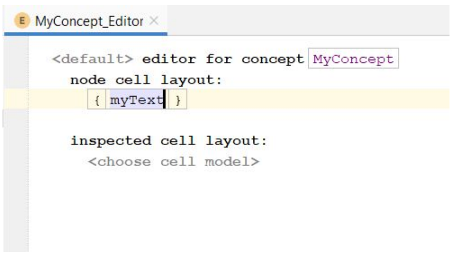
Then, you should create an Editor for your concept so that it can be nicely displayed to your users. Push the plus button on the bottom of the window (Figure 4.15) and select *Editor* and then *Concept Editor*.



```
concept MyConcept extends    BaseConcept
                  implements <none>

  instance can be root: false
  alias: <no alias>
  short description: <no short description>

  properties:
  myText : string

  children:
  << ... >>

  references:
  << ... >>
```

Figure 4.15: Adding a new aspect for a concept

In the opened tab, go with your cursor to the red rectangle and hit *CTRL+Space* shortcut and select *{myText}*, just as in Figure 4.16.



```
<default> editor for concept MyConcept
  node cell layout:
    { myText }

  inspected cell layout:
    <choose cell model>
```

Figure 4.16: Initializing the first concept's editor

Then, you will create the Generator aspect for your concept. This will serve to transform AST nodes of your concept into chunks of code of the C# base language. Push the plus button again, select *Generator* and the *Template Declaration*.

Now, you will add a dependency on the C# base language so that you will be able to use it for the above-mentioned chunks of code.

Expand the *generator/MyLanguage/main* item in the project explorer, right-click the *main@generator* item and select *Model Properties* (see Figure 4.17).

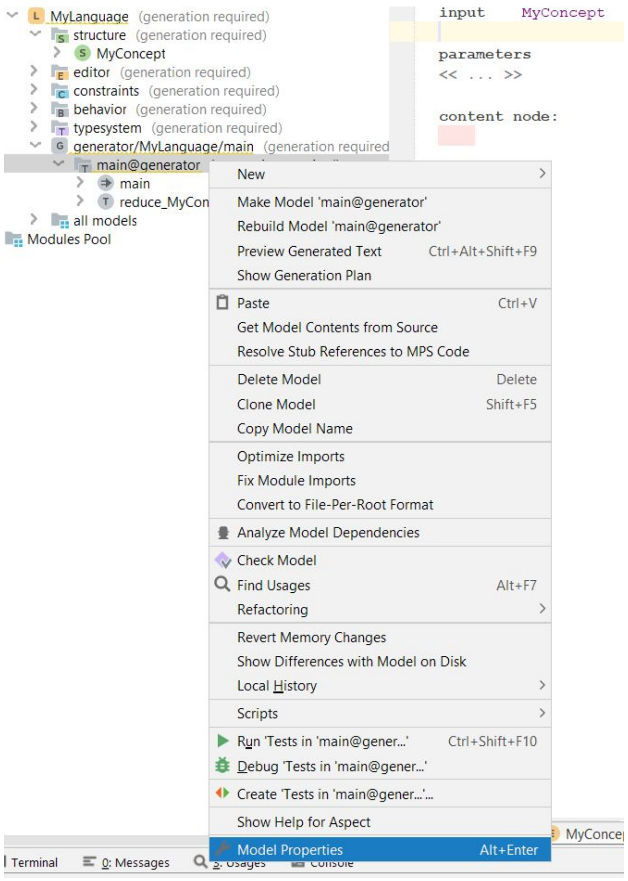
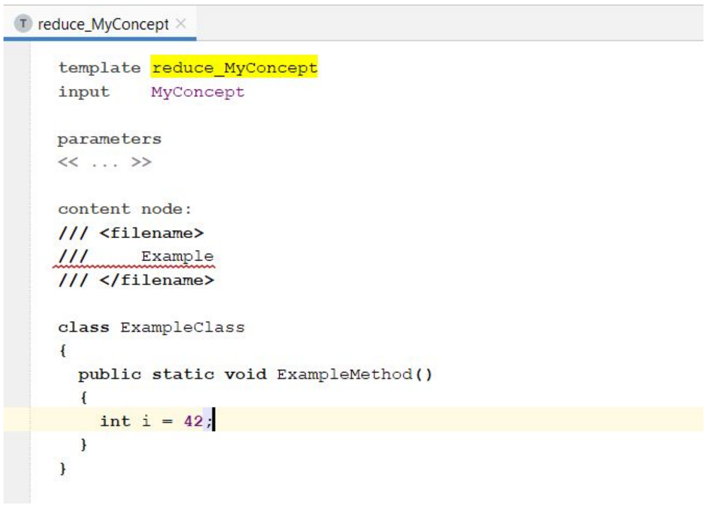

MyLanguage (generation required)
structure (generation required)
MyConcept
editor (generation required)
constraints (generation required)
behavior (generation required)
typesystem (generation required)
generator/MyLanguage/main (generation required
main@generator
main
reduce_MyCon
all models
Modules Pool
input MyConcept
parameters
<< ... >>
content node:
New
Make Model 'main@generator'
Rebuild Model 'main@generator'
Preview Generated Text Ctrl+Alt+Shift+F9
Show Generation Plan
Paste Ctrl+V
Get Model Contents from Source
Resolve Stub References to MPS Code
Delete Model Delete
Clone Model Shift+F5
Copy Model Name
Optimize Imports
Fix Module Imports
Convert to File-Per-Root Format
Analyze Model Dependencies
Check Model
Find Usages Alt+F7
Refactoring
Revert Memory Changes
Show Differences with Model on Disk
Local History
Scripts
Run 'Tests in 'main@gener...' Ctrl+Shift+F10
Debug 'Tests in 'main@gener...'
Create 'Tests in 'main@gener...'...
Show Help for Aspect
Model Properties Alt+Enter
MyConce
Terminal
0: Messages

Figure 4.17: Setting dependencies for Generator

In the opened window, under the tab *Used languages*, push the plus button on the right. Type in *CsBaseLanguage* and hit *Enter*. You should get the result presented by Figure 4.6 from the previous Section *4.1*. By this, you have added the C# base language among the languages that you can use in the Generator aspect.

Then, under the tab *Dependencies*, push the plus button again, type in *System* and select the found item. Then you should see what is illustrated by Figure 4.7 in the previous section *4.1*. This has made available the *System* namespace of the C# standard library.

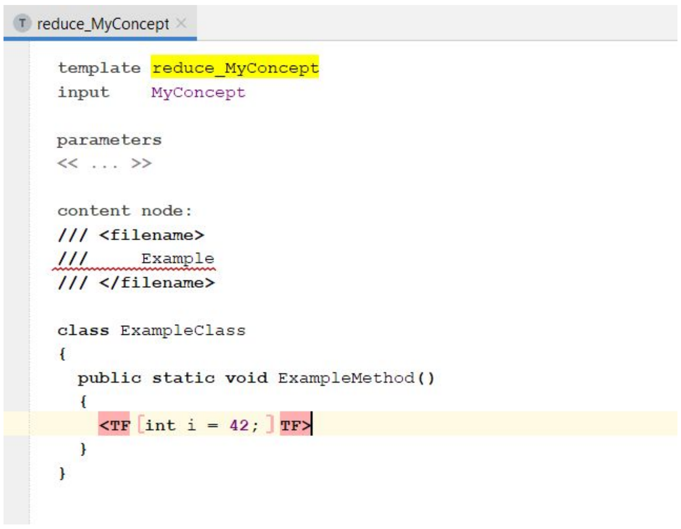
Now, return to your Generator for your concept. Put cursor in the red rectangle and start creating a C# base language program. Start with a *File* AST node and then for example, create what is illustrated in Figure 4.18. Do not worry about the red underlined text, you will fix it in a while.



```
reduce_MyConcept

template reduce_MyConcept
input    MyConcept

parameters
<< ... >>

content node:
/// <filename>
///      Example
/// </filename>

class ExampleClass
{
  public static void ExampleMethod()
  {
    int i = 42;
  }
}
```

Figure 4.18: Basics of the created Generator

Put your cursor as in Figure 4.18, hit *CTRL* together with the up arrow twice. Then hit *Alt+Enter* and select *Create Template Fragment*. You should obtain what is shown in Figure 4.19.



```
reduce_MyConcept

template reduce_MyConcept
input    MyConcept

parameters
<< ... >>

content node:
/// <filename>
///      Example
/// </filename>

class ExampleClass
{
  public static void ExampleMethod()
  {
    <TF [int i = 42; ] TF>
  }
}
```

Figure 4.19: Template Fragment

The code wrapped in the template fragment, i.e. in the marked area, is the chunk of C# base language code that will be used whenever an AST node of your concept will be transformed into C# base language. If you define such Generators for all the concept in your DSL in such a way that they will fit together, the result of the transformation of a program written in your DSL into C# base language will be a valid C# base language program.

Of course, the DSL created in this tutorial is unusable, but the information you learned here should be enough to get you started using our C# base language for DSL development. To develop more complex DSLs, you will probably have to dive into the MPS documentation.