Working with EMF model editor:

1. To start working on EMF models, create a new Empty EMF project (Eclipse Modeling Framework section). Then, add to the model folder a new Ecore diagram (right click on the model folder). Two files will be created:
   a. .ecore, which contains model itself
   b. .ecorediag, which is a view of model expressed using UML notation

Theoretically, model and its view are supposed to be always synchronized, but in practice you can encounter some synchronization problems when you work concurrently on these 2 files. Therefore I would recommend you to work either on .ecore or .ecorediag (which is probably more reasonable choice) file, and avoid having both files opened at the same time.

2. In the outline view, you can find all classes defined in your .ecore model. Note, that if you delete a class from diagram using ‘del’ button, the class will be removed from the diagram only. In other words, it will be still a part of the model (visible in the outline view). After such an operation, you can add the class again to the diagram by ‘drag & drop’ action from the outline view.

If you want to remove the class permanently from the model, right click on the class and choose ‘Remove from model’ action. You can remove the class from the model using outline view as well.

3. If you select a type (e.g. the type of attribute in a class) sometimes it happens that there are no entries in the list to choose from. To go around this bug, you can mark ‘case sensitive’ field (in case when was not marked) or unmark it (when it was marked) and click ok. The list should be available when you open the type selection dialog one more time.

4. It’s a good idea to validate your model before saving and generating a code (right click on .ecore model and click ‘Validate’). If there are some errors indicated, you can click on ‘Details’ to see the description, and double click on description to see the cause of error highlighted.

Code generation:

1. To generate the code, you have to create a new ‘EMF generator model’ (in the wizard choose your .core file and click load). Then, from newly created .genmodel file you can generate (right click) model code, edit code, editor code.

2. If you want to regenerate the code having done some changes in the model, you have to update .genmodel by right click on it and click reload. To protect the code you have changed (e.g when you implemented the body of some method) from being overwritten while regenerating the code, change the annotation from ‘@generated’ to ‘@generated NOT’
**Model code:**

Generating a model code results in creating 3 packages in your project (src):

- ModelName – where you can find:
  - interfaces of the classes in your model generated
    Note, that for each attribute of a class there are getters and setters automatically generated.
  - interface of a factory class, which you should use for creating new instances of your classes. To access the factory static instance you should use something like FactoryName.eINSTANCE
  - interface of a package class, which provides some static constants (for example, the feature constants used by the generated methods) and convenience methods for accessing your model's metadata

- ModelName.impl – where you can find implementation of the interfaces defined in the package above.
  - Note, that implementation of the generated getters and setters enables e.g. installing observers on class instances (notification mechanism) as well as synchronization in case of two-way references (if you change on side of two-way reference, the other is automatically set as well)
  - If a reference has cardinality 0 or 1, normal getters and setters are generated. Otherwise (when upper bound is equal to ‘*’), there is no setter generated. This is because ‘*’ means simply some collection of objects, and there is no point in setting this collection, because we want to set rather some particular elements of the collection. Thus, the getter returns an Elist (which can be used exactly as java.util.List), and we can add elements to this list or remove them from the list as well (which is conceptually an equivalent of setting).

- ModelName.util - which you won’t probably need;)

For details, take a look at:


**Edit code:**

Generating an edit code results in creating a new plug-in project, with .edit at the end of project name.

This is a short overview of a EMF.Edit framework taken from eclipse help website:

EMF.Edit is an Eclipse framework that includes generic reusable classes for building editors for EMF models. It provides:

- Content and label provider classes, property source support, and other convenience classes that allow EMF models to be displayed using standard desktop (JFace) viewers and property sheets.
• A command framework, including a set of generic command implementation classes for building editors that support fully automatic undo and redo.

• A code generator capable of generating everything needed to build a complete editor plug-in for your EMF model. It produces a properly structured editor that conforms to the recommended style for Eclipse EMF model editors. You can then customize the code however you like, without losing your connection to the model.

All the relevant details concerning EMF.edit will be explained later on (next tutorials). Anyway, you can find a very good description here:


Editor code:
Generating an editor code results in creating a new plug-in project, with .editor at the end of project name. This plug-in implements a tree editor for your model (accessible in a run-time workbench) and you won’t probably be changing anything in it.