Packages

Packages for Avoiding Name Collisions

- Modelica provide a safe and systematic way of avoiding name collisions through the package concept
- A package is simply a container or name space for names of classes, functions, constants and other allowed definitions

Packages as Abstract Data Type: Data and Operations in the Same Place

Keywords denoting a package

encapsulated makes package dependencies (i.e., imports) explicit

Declarations of substract, divide, realPart, imaginaryPart, etc are not shown here

```
encapsulated package ComplexNumber

  record Complex
    Real re;
    Real im;
  end Complex;

  function add
    input  Complex x,y;
    output Complex z;
  algorithm
    z.re := x.re + y.re;
    z.im := x.im + y.im
  end add;

  function multiply
    input  Complex x,y;
    output Complex z;
  algorithm
    z.re := x.re*y.re – x.im*y.im;
    z.im := x.re*y.im + x.im*y.re;
  end multiply;
  ……..
end ComplexMumbers
```

Usage of the `ComplexNumber` package

```
class ComplexUser
  ComplexNumbers.Complex  a(re=1.0, im=2.0);
  ComplexNumbers.Complex  b(re=1.0, im=2.0);
  ComplexNumbers.Complex  z,w;
equation
  z = ComplexNumbers.multiply(a,b);
  w = ComplexNumbers.add(a,b);
end ComplexUser
```

The type `Complex` and the operations `multiply` and `add` are referenced by prefixing them with the package name `ComplexNumber`

Accessing Definitions in Packages

- Access reference by prefixing the package name to definition names

```
class ComplexUser
  ComplexNumbers.Complex  a(re=1.0, im=2.0);
  ComplexNumbers.Complex  b(re=1.0, im=2.0);
  ComplexNumbers.Complex  z,w;
equation
  z = ComplexNumbers.multiply(a,b);
  w = ComplexNumbers.add(a,b);
end ComplexUser
```

- Shorter access names (e.g. `Complex`, `multiply`) can be used if definitions are first imported from a package (see next page).

Importing Definitions from Packages

- Qualified import ← `import <packagename>`
- Single definition import ← `import <packagename> . <definitionname>`
- Unqualified import ← `import <packagename> . *`
- Renaming import ← `import <shortpackagename> = <packagename>`

The four forms of import are exemplified below assuming that we want to access the addition operation (`add`) of the package `Modelica.Math.ComplexNumbers`

```
import Modelica.Math.ComplexNumbers;      //Access as ComplexNumbers.add
import Modelica.Math.ComplexNumbers.add;  //Access as add
import Modelica.Math.ComplexNumbers.*     //Access as add
import Co = Modelica.Math.ComplexNumbers  //Access as Co.add
```


Qualified Import

Qualified import ← `import <packagename>`

The *qualified import* statement
`import` <*packagename*>`;`
imports all definitions in a package, which subsequently can be referred to by (usually shorter) names
`simplepackagename . definitionname`, where the simple package name is the *`packagename`* without its prefix**.**

```
encapsulated package ComplexUser1
  import Modelica.Math.ComplexNumbers;
  class User
    ComplexNumbers.Complex  a(x=1.0, y=2.0);
    ComplexNumbers.Complex  b(x=1.0, y=2.0);
    ComplexNumbers.Complex  z,w;
  equation
    z = ComplexNumbers.multiply(a,b);
    w = ComplexNumbers.add(a,b);
  end User;
end ComplexUser1;
```

This is the most common form of `import` that eliminates the risk for name collisions when importing from several packages

Single Definition Import

Single definition import ← `import <packagename> . <definitionname>`

The *single definition import* of the form
`import <packagename>.<definitionname>;`
allows us to import a single specific definition (a constant or class but not a subpackage) from a package and use that definition referred to by its *definitionname* without the package prefix

```
encapsulated package ComplexUser2
  import ComplexNumbers.Complex;
  import ComplexNumbers.multiply;
  import ComplexNumbers.add;
 class User
  Complex  a(x=1.0, y=2.0);
  Complex  b(x=1.0, y=2.0);
  Complex  z,w;
 equation
  z = multiply(a,b);
  w = add(a,b);
 end User;
end ComplexUser2;
```

There is no risk for name collision as long as we do not try to import two definitions with the same short name

Unqualified Import

Unqualified import ← `import <packagename> . *`

The unqualified import statement of the form
`import packagename.*;`
imports all definitions from the package using their short names without qualification prefixes.
Danger: Can give rise to name collisions if imported package is changed.

```
class ComplexUser3
  import ComplexNumbers.*;
  Complex  a(x=1.0, y=2.0);
  Complex  b(x=1.0, y=2.0);
  Complex  z,w;
equation
  z = multiply(a,b);
  w = add(a,b);
end ComplexUser3;
```

This example also shows direct `import` into a class instead of into an enclosing package

Renaming Import

Renaming import ← `import <shortpackagename> = <packagename>`

The *renaming import* statement of the form:
`import` <*`shortpackagename`*> `=` <*`packagename`*>`;`
imports a package and renames it locally to *`shortpackagename`*. One can refer to imported definitions using *`shortpackagename`* as a presumably shorter package prefix**.**

```
class ComplexUser4
  import Co = ComplexNumbers;
  Co.Complex  a(x=1.0, y=2.0);
  Co.Complex  b(x=1.0, y=2.0);
  Co.Complex  z,w;
equation
  z = Co.multiply(a,b);
  w = Co.add(a,b);
end ComplexUser4;
```

This is as safe as qualified import but gives more concise code

Package and Library Structuring

A well-designed package structure is one of the most important aspects that influences the complexity, understandability, and maintainability of large software systems. There are many factors to consider when designing a package, e.g.:

- The name of the package.
- Structuring of the package into subpackages.
- Reusability and encapsulation of the package.
- Dependencies on other packages.

Subpackages and Hierarchical Libraries

The main use for Modelica packages and subpackages is to structure hierarchical model libraries, of which the standard Modelica library is a good example.

```
encapsulated package Modelica         // Modelica
   encapsulated package Mechanics     // Modelica.Mechanics
      encapsulated package Rotational // Modelica.Mechanics.Rotational
          model Inertia // Modelica.Mechanics.Rotational.Inertia
           ...
          end Inertia;
          model Torque  // Modelica.Mechanics.Rotational.Torque
           ...
          end Torque;
          ...
     end Rotational;
   ...
end Mechanics;
...
end Modelica;
```


Ecapsulated Packages and Classes

An encapsulated package or class *prevents* direct reference to public definitions *outside* itself, but as usual allows access to public subpackages and classes inside itself.

- Dependencies on other packages become explicit
  – more readable and understandable models!
- Used packages from outside must be *imported*.

```
encapsulated model TorqueUserExample1
   import Modelica.Mechanics.Rotational;  // Import package Rotational
   Rotational.Torque  t2;                 // Use Torque, OK!
   Modelica.Mechanics.Rotational.Inertia  w2;
        //Error! No direct reference to the top-level Modelica package
   ...   // to outside an encapsulated class
end TorqueUserExample1;
```


`within` Declaration for Package Placement

Use *short names* without dots when declaring the package or class in question, e.g. on a separate file or storage unit. Use `within` to specify within which package it is to be placed.

The within declaration states the *prefix* needed to form the fully qualified name

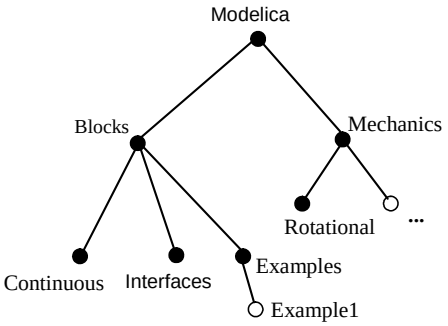
```
within Modelica.Mechanics;
encapsulated package Rotational // Modelica.Mechanics.Rotational
  encapsulated package Interfaces
     import ...;
     connector Flange_a;
        ...
     end Flange_a;
     ...
  end Interfaces;
  model Inertia
     ...
  end Inertia;
  ...
end Rotational;
```

The subpackage `Rotational` declared within `Modelica.Mechanics` has the fully qualified name `Modelica.Mechanics.Rotational`, by concatenating the *packageprefix* with the *short name* of the package.

Mapping a Package Hierachy into a Directory Hirarchy

A Modelica package hierarchy can be mapped into a corresponding directory hierarchy in the file system

```
C:\library
     \Modelica
         package.mo
         \Blocks
            package.mo
            Continuous.mo
            Interfaces.mo
            \Examples
              package.mo
              Example1.mo
          \Mechanics
            package.mo
            Rotational.mo
            ...
```



Mapping a Package Hierachy into a Directory Hirarchy