#### 8.2.3.1 General format of import

An import statement can be used anywhere in the module definitions part.

***Syntactical Structure***

[ *Visibility* ] **import** **from** *ModuleId*

 (

 ( **all** [ **except** "{" *ExceptSpec* "}" ] )

 |

 ( "{" *ImportSpec* "}" )

 )

[ ";" ]

***Semantic Description***

TTCN‑3 supports the import of the following definitions: module parameters, user defined types, signatures, constants, data templates, signature templates, functions, external functions, altsteps and test cases. Each definition has a *name* (defines the identifier of the definition, e.g. a function name), a *specification* (e.g. a type specification or a signature of a function) and in the case of functions, altsteps and test cases an associated *behaviour description*. In addition, import statements of one module can be explicitly imported by another module (see clause 8.2.3.7). Only definitions or import statements visible from the importing module can be imported (see clause 8.2.5).

In contrast to module definitions, which are by default public, import statements are by default private.

EXAMPLE 1a:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Name | Specification | Behaviour description |
| **function** | MyFunction | (**inout** MyType1 MyPar) **return** MyType2**runs on** MyCompType | { const MyType3 MyConst := …; : // further behaviour} |

|  |  |  |  |
| --- | --- | --- | --- |
|  | Specification | Name | Specification |
| **type** | **record** | MyRecordType | { field1 MyType4, field2 integer} |

|  |  |  |  |
| --- | --- | --- | --- |
|  | Specification | Name | Specification |
| **template** | MyType5 | MyTemplate | := { field1 := 1, field2 := MyConst, // MyConst is a module constant field3 := ModulePar // ModulePar is module parameter} |

Behaviour descriptions have no effect on the import mechanism, because their internals are considered to be invisible to the importer when the corresponding functions, altsteps or test cases are imported. Thus, they are not considered in the following descriptions.

The specification part of an importable definition contains *local definitions* (e.g. field names of structured type definitions or values of enumerated types) and *referenced definitions* (e.g. references to type definitions, templates, constants or module parameters). For the examples above, this means:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Name | Local definitions | Referenced definitions |
| **function** | MyFunction | MyPar | MyType1, MyType2, MyCompType |
| **type** | MyRecordType | field1, field2 | MyType4, integer |
| **template** | MyTemplate |  | MyType5, field1, field2, field3, MyConst, ModulePar |

NOTE 1: The local definitions column refers to identifiers only that are newly defined in the importable definition. Values assigned to individual fields of importable definitions, e.g. in template definitions, may also be considered as local definitions, but they are not important for the explanation of the import mechanism.

NOTE 2: The referenced definitions field1, field2 and field3 of template MyTemplate are the field names of MyType5, i.e. they are referenced via MyType5.

Referenced definitions are also importable definitions, i.e. the source of a referenced definition can again be structured into a name and a specification part and the specification part also contains local and referenced definitions. In other words, an importable definition may be built up recursively from other importable definitions.

The TTCN‑3 import mechanism is related to the local and referenced definitions used in the specification part of the importable definitions. Table 8 specifies the possible local and referenced definitions of importable definitions.

Table 8: Possible local and referenced definitions of importable definitions

|  |  |  |
| --- | --- | --- |
| Importable Definition | Possible Local Definitions | Possible Referenced Definitions |
| Module parameter |  | Module parameter type |
| User-defined type (for all) |  |  |
| * enumerated type
 | Concrete values |  |
| * structured type
 | Field names, nested type definitions | Field types |
| * port type
 |  | Message types, signatures |
| * component type
 | Constant names, variable names, timer names and port names | Constant types, variable types, port types |
| Signature | Parameter names | Parameter types, return type, types of exceptions |
| Constant |  | Constant type |
| Data Template | Parameter names | Template type, parameter types, constants, module parameters, functions |
| Signature template |  | Signature definition, constants, module parameters functions |
| Function | Parameter names | Parameter types, return type, component type (**runs on** clause) |
| External function | Parameter names | Parameter types, return type |
| Altstep | Parameter names | Parameter types, component type (**runson** clause) |
| Test case | Parameter names | Parameter types, component types (**runs on**- and **system** clause) |
| NOTE 1: For the import of import statements see clause 8.2.3.7.NOTE 2: For the import of groups see clause 8.2.3.3. |

The TTCN‑3 import mechanism distinguishes between the *identifier of a referenced definition* and the *information necessary for the usage of a referenced definition* within the imported definition. For the usage, the identifier of a referenced definition is not required and therefore not imported automatically.

EXAMPLE 1b: Differentiation between *information necessary for the usage* and the identifier.

 **module** A {

 **type** **record** MyRec1 {

 **integer** field1,

 **charstring** field2

 }

 }

 **module** B {

 **import** **from** A **all**;

 **type** **record** MyRec2 {

 MyRec1 myField1,

 // "myField1" is the local definition, "MyRec1" is a referenced definition;

 // the *name* "MyRec1" shall be imported in this case as is directly referenced

 **boolean** myField2

 }

 }

 **module** C {

 **import** **from** B **all**;

 **const** MyRec2 t\_MyRec2 := {

 myField1 := { field1 := 5, field2 := "A" },

 // to define myField1 of MyRec2 the name "MyRec1" is not needed, the

 // *information necessary for the usage* is its type information,

 // i.e. names and types of its fields field1 and field2

 // which is embeddded in the imported definition of MyRec2

 myField2 := **true**

 }

 }

If an imported definition has attributes (defined by means of a **with** statement) then the attributes shall also be imported. The mechanism to change attributes of imported definitions is explained in clause 27.1.3.

NOTE 3: If the module has global attributes they are associated to definitions without these attributes.

The use of **import** on single definitions, groups of definitions, definitions of the same kind, etc. may lead to situations where the *same definition is referred to more than once*. Such cases shall be resolved by the system and definitions shall be imported only once.

NOTE 4: The mechanisms to resolve such ambiguities, e.g. overwriting and sending warnings to the user, are outside the scope of the present document and should be provided by TTCN‑3 tools.

All **import** statements and definitions within import statements are considered to be treated independently one after the other in the order of their appearance.

All TTCN‑3 modules shall have their own name space in which all definitions shall be uniquely identified. *Name clashes* may occur due to import, e.g. import from different modules. Name clashes shall be resolved using qualified name(s) for the imported definition(s), i.e. prefixing the imported definition (which causes the name clash) by the identifier of the module in which it has been defined; the prefix and the identifier shall be separated by a dot (".").

There is one exception to this rule: when **in the context** of an enumerated type (see clause 6.2.4), an enumerated value is clashing with the name of a definition in the importing module, the enumerated value shall take precedence and the definition in the importing module shall be referenced by using its qualified name (see example 4 below in this clause).

In cases where there are no ambiguities the prefixing need not (but may) be present when the imported definitions are used. When the definition is referenced in the same module where it is defined, the module identifier of the module (the current module) also may be used for prefixing the identifier of the definition. For the latter case, prefixing shall only be used for definitions with global visibility for the module.

***Restrictions***

In addition to the general static rules of TTCN‑3 given in clause 5, the following restrictions apply:

a) An import statement shall only be used in the module definitions part and not be used within a control part, function definition, and alike.

b) Only top-level visible definitions of a module may be imported. Definitions which are top-level but invisible to the importing module or which occur at a lower scope (e.g. local constants defined in a function) shall not be imported.

c) A definition is imported together with its name and all local definitions.

NOTE 5: A local definition, e.g. a field name of a user-defined record type or an enumerated value, has only meaning in the context of the definitions in which it is defined, e.g. a field name of a record type can only be used to access a field of the record type and not outside this context.

 In particular, importing an enumerated type does not impose the restriction given in clause 6.2.4 on global names defined in the importing module.

d) A definition is imported together with all information of referenced definitions that are necessary for the usage of the imported definition, independent of the visibility of the referenced definitions (see clause 8.2.5).

NOTE 6: If module C imports a definition from module B that uses a type reference defined in module A, the corresponding information necessary for the usage of that type is automatically imported into module C (see example 5 below in this clause). Identifiers of referenced definitions are not automatically imported.

 In particular, if module C imports global value or template definitions (e.g. constants, module parameters, templates) or local definitions (e.g. formal parameters of templates, functions, etc., or constants and variables of component types) of an enumerated type from module B, the enumerated values of this type (i.e. the identifiers) are implicitly and automatically imported to module C. That is, the enumerated values are known when an enumerated value or template is used in module C (e.g. when an actual parameter is passed or a value is assigned to a component variable). Note that this implicit importing does not impose the restriction given in clause 6.2.4 on global names defined in module C.

e) If the referenced definitions are wished to be used in the importing module, they shall be explicitly imported either directly from its source module or indirectly by importing the import statements of a module importing it (see clause 8.2.3.7).

f) When importing a function, altstep or test case the corresponding behaviour specifications and all definitions used inside the behaviour specifications remain invisible for the importing module.

g) The language specification (see clause 8.1) of the import statement shall not override the language specification of the importing module.

h) The language specification of the import statement shall be identical to the language specification of the source module from which definitions are imported (see clause 8.2.3.8) provided a language specification is defined in the source module. If not, the language specification in the import statement is taken as the language specification of the source module. If the source module uses however language concepts not being part of that language specification, this causes an error for the import statement.

***Examples***

EXAMPLE 1: Selected import examples

**module** MyModuleA

{ :

 // Scope of the imported definitions is global to MyModuleA

 **import** **from** MyModuleB **all**; // import of all definitions from MyModuleB

 **import from** MyModuleC { // import of selected definitions from MyModuleC

 **type** MyType1, MyType2; // import of types MyType1 and MyType2

 **template** **all** // import of all templates

}

 :

  **function** MyBehaviourC()

 {

 // import cannot be used here

 :

 }

 **:**

 **control**

{

 // import cannot be used here

:

}

}

EXAMPLE 2: Use of imported definitions and visibility of definitions referenced by them

 **module** ModuleONE {

 **modulepar** **integer** ModPar1 := …;

 **type** **record** RecordType\_T1 {

 **integer** Field1\_T1,

  **:**

 }

 **type** **record** RecordType\_T2 {

 RecordType\_T1 Field1\_T2,

 :

 }

 **const** **integer** MyConst := …;

 **template** RecordType\_T2 Template\_T2 (RecordType\_T1 TempPar\_T2):= { // parameterized template

 Field1\_T2 := …,

 :

 }

 } // end module ModuleONE

 **module** ModuleTWO {

 **import** **from** ModuleONE {

 **template** Template\_T2

 }

 // Only the names Template\_T2 and TempPar\_T2 will be visible in ModuleTWO. Please note, that

 // the identifier TempPar\_T2 can only be used when modifying Template\_T2. All information

 // necessary for the usage of Template\_T2, e.g. for type checking purposes, are imported

 // for the referenced definitions RecordType\_T1, Field1\_T2, etc., but their identifiers are

 // not visible in ModuleTWO.

 // This means, e.g. it is not possible to use the constant MyConst or to declare a

 // variable of type RecordType\_T1 or RecordType\_T2 in ModuleTWO without explicitly importing
 // these types.

 **import** **from** ModuleONE {

 **modulepar** ModPar2

 }

 // The module parameter ModPar2 of ModuleONE is imported from ModuleONE and

 // can be used like an integer constant

 } // end module ModuleTWO

 **module** ModuleTHREE {

 **import** **from** ModuleONE **all**; // imports all definitions from ModuleONE

 **type port** MyPortType **message** {

 **inout** RecordType\_T2 // Reference to a type defined in ModuleONE

 }

 **type component** MyCompType {

 **var integer** MyComponentVar := ModPar2;

 // Reference to a module parameter of ModuleONE

 **:**

 }

 **function** MyFunction () **return** **integer** {

 **return** MyConst // Reference to a module constant of ModuleONE

 }

 **testcase** MyTestCase (**out** RecordType\_T2 MyPar) **runs** **on** MyCompType {

 :

 MyPort.**send**(Template\_T2); // Sending a template defined in ModuleONE

 :

 }

 } // end ModuleTHREE

 **module** ModuleFOUR {

 **import from** ModuleTHREE {

 **testcase** MyTestCase

 }

 // Only the name MyTestCase will be visible and usable in ModuleFOUR.

 // Type information for RecordType\_T2 is imported via ModuleTHREE from ModuleONE and

 // Type information for MyCompType is imported from ModuleTHREE. All definitions

 // used in the behaviour part of MyTestCase remain hidden for the user of ModuleFOUR.

 } // end ModuleFOUR

EXAMPLE 3: Handling of name clashes

 **module** MyModuleA {

 :

 **type** **bitstring** MyTypeA;

 **import from** SomeModuleC {

 **type** MyTypeA, // Where MyTypeA is of type character string

 MyTypeB // Where MyTypeB is of type character string

 }

 **:**

 **control** {

 :

 **var** SomeModuleC.MyTypeA MyVar1 := "Test String"; // Prefix shall be used

 **var** MyTypeA MyVar2 := '10110011'B; // This is the original MyTypeA

 :

 **var** MyTypeB MyVar3 := "Test String"; // Prefix need not be used …

 **var** SomeModuleC.MyTypeB MyVar3 := "Test String"; // … but it can be if wished

 :

}

 }

NOTE 7: Definitions with the same name defined in different modules are always assumed to be different, even if the actual definitions in the different modules are identical. For example, importing a type that is already defined locally, even with the same name, would lead to two different types being available in the module.

EXAMPLE 4: Name clash between enumerated values and global definitions

 **module** A {

 **type** **enumerated** MyEnumType {enumX, enumY, enumZ}

 **type** **enumerated** MyEnumType2 {enumX, enumY, enumZ}

 }

 **module** B {

 **import** **from** A **all**;

 **const** MyEnumType enumY := enumX; // this is not allowed as enumerated values restrict

 // global names (see clause 6.2.4)

 **const** MyEnumType2 enumX := enumX;// this is likewise not allowed

 **const** **integer** enumZ := 0;

 **modulepar** MyEnumType px\_MyModulePar1 := enumY

 // the default value of the module parameter will be the value enumY, as the type of

 // px\_MyModulePar1 creates the context of MyEnumType and in this context enumerated values

 // take precedence over global definition names; note that for the same context reason there

 // in no name clash between the enumerated values defined in MyEnumType and in MyEnumType2

 **modulepar** MyEnumType px\_MyModulePar2 := B.enumY

 // the default value of the module parameter will be the value enumX, as the prefix

 // identifies the constant definition enumY unambiguously, which has the value enumX

 **modulepar** **integer** px\_IntegerPar := enumZ;

 // the default value of the module parameter will be 0 as this assignment is not in the

 // context of an enumerated type, hence no name clash occurs

 **modulepar** MyEnumType px\_MyModulePar3 := B.enumX

 // causes an error as px\_MyModulePar3 and the constant enumX has different types

 }

EXAMPLE 5: Importing local definitions transitively

 **module** A {

 **type enumerated** MyEnum\_Type { enumX, enumY, enumZ}

 **type record** MyRec { **integer** a, **integer** b }

 **type component** MyComp { **var** MyRec v\_Rec := { a := 5 } }

 }

 **module** B {

 **import** **from** A **all**;

 **modulepar** MyEnum\_Type px\_MyModulePar := enumY;

 **type component** MyCompUser **extends** MyComp {}

 }

 **module** C {

 **import from** B **all**;

 **testcase** TC() **runs on** MyCompUser {

 **if** (px\_MyModulePar == enumY) {

 // the enumerated value enumY is know in C without explicitly importing it from A

 **setverdict**(**pass**)

 }

 **if** (v\_Rec.a == 5) {

 v\_Rec.b := v\_Rec.a;

 // Both the variable name v\_Rec and the record field names are known in C without

 // explicitly importing them from A

 **setverdict** (**pass**)

 }

 }

 }

# Annex G (informative):Deprecated language features

# G.10 Prefixing identifiers of local definitions with module identifiers

Previous versions of the present document (up to and including V4.6.1) do not exclude the possibility to prefix identifier of definitions without global visibility (e.g. templates defined in functions or test cases) with the local module identifier. Prefixing identifiers of local definitions with module identifiers is deprecated and may be fully removed in a future edition of the present document.