

26 Encapsulation Constructs

This chapter contains syntax and descriptions of the *e* statements that are used to create packages and modify access control. The constructs are:

- **package package-name** on page 26-1
- **package type-declaration** on page 26-2
- **package | protected | private struct-member** on page 26-4
- “Scope Operator” on page 26-6

Note See also Chapter 3 “Constant Fields and Constant when Subtypes” in *Specman Beta Features*

26.1 package package-name

Purpose

Associates a module with a package.

Category

Statement

Syntax

package *package-name*

Syntax Example

```
package vr_xb;
```

Parameters

<i>package-name</i>	A standard <i>e</i> identifier which assigns a unique name to the package. It is legal for a package name to be the same as a module or type name.
---------------------	--

Description

Only one **package** statement can appear in a file, and it must be the first statement in the file.

A file with no **package** statement is equivalent to a file beginning with the statement, **package main**.

Example

```
<'
// module vr_xb_top
package vr_xb;
'>
```

See Also

- [package type-declaration](#) on page 26-2
- [package | protected | private struct-member](#) on page 26-4
- [Chapter 9 “Encapsulation in e”](#) in the *Usage and Concepts Guide for e Testbenches*

26.2 package type-declaration

Purpose

Modifies access to a type or a struct.

Category

Statement

Syntax

[package] *type-declaration*

Syntax Example

```
package type t: int(bits: 16);
```

Parameters

type-declaration An *e* type declaration (for a struct, unit, enumerated list, or other type).

Description

The **package** modifier means that code outside the package files cannot access the defined struct member. This includes declaring a variable of the type, extending, inheriting, casting using the **as_a** operator, and all other contexts in which the name of a type is used. It is equivalent to the default (package) access level for classes in Java.

Note The package type does not determine the visibility of a package, but only its access control.

Without the **package** modifier, the type or struct has no access restriction.

Definition of a when subtype (using a **when** or **extend** clause) does not allow for an access modifier. A when subtype is public unless either its base struct or one of its determinant fields is declared **package**.

A **when** subtype cannot have a **private** or **protected** determinant field. Any reference to a **when** subtype, even in a context in which the **when** determinant field is accessible, results in a compilation error.

Similarly, a keyed list is public unless either its element type or its key field is declared as a **package**. A keyed list cannot have **private** or **protected** key field. Reference to such a list results in a compilation error.

Example

```
<'
// module vr_xb_top
package vr_xb;;
package type width: uint(bits: 8);
'>
```

See Also

- [package package-name](#) on page 26-1
- [package | protected | private struct-member](#) on page 26-4
- Chapter 9 “Encapsulation in e” in the *Usage and Concepts Guide for e Testbenches*

26.3 package | protected | private struct-member

Purpose

Modifies access to a struct field, method, or event.

Category

Keyword

Syntax

package *struct-member-definition*

protected *struct-member-definition*

private *struct-member-definition*

Syntax examples:

```
private f: int;  
protected m() is {};  
package event e;
```

Parameters

<i>struct-member-definition</i>	A struct or unit field, method, or event definition. See “Structs, Fields and Subtypes” in the <i>e Language Reference</i> for the syntax of struct and unit member definitions.
---------------------------------	--

Description

A struct member declaration may include a **package**, **protected**, or **private** keyword to modify access to the struct member.

If no access modifier exists in the declaration of a struct member, the struct member has no access restriction (the default is public).

The **package** modifier means that code outside the package files cannot access the struct member. It is equivalent to the default (package) access level for fields and methods in Java.

The **protected** modifier means that code outside the struct family scope cannot access the struct member. It is similar (although not equivalent) to the *protected* semantics in other object-oriented languages.

The **private** modifier means that only code within both the package and the struct family scope can access the struct member. This means that code within the extension of the same struct in a different package is outside its accessibility scope. It is less restrictive than *private* attribute of other object-oriented languages in the sense that methods of derived structs or units within the same package can access a private struct member.

An extension of a struct member can restate the same access modifier as the declaration has, or omit the modifier altogether. If a different modifier appears, the compiler issues an error.

All references to a struct member outside its accessibility scope result in an error at compile time. Using an enumerated field's value as a **when** determinant is considered such a reference, even if the field name is not explicitly mentioned.

A field must be declared **package** or **private** if its type is **package**.

A method must be declared **package** or **private** if its return type or any of its parameter types are **package**.

Only fields, methods and events can have access restrictions. There are other named struct members in *e*, namely cover groups and named expects, to which access control does not apply - they are completely public. However, cover groups and expects are defined in terms of fields, methods and events, and can refer to other entities in their definitions according to the accessibility rules.

Example

```
<'
package P1;

struct s1 {
    private f: int;
    protected m() is {};
    package event e;
};
'>
```

See Also

- [package package-name](#) on page 26-1
- [package type-declaration](#) on page 26-2
- Chapter 9 “Encapsulation in *e*” in the *Usage and Concepts Guide for e Testbenches*

26.4 Scope Operator

26.4.1 ::

Purpose

Identify the package scope for the given type reference

Category

Special purpose operator

Syntax

package-name :: *type-name*

Syntax Example

```
xbus_evc: vr_xbus::env_u;
```

Parameters

package-name The name of the package in which the type was declared. The * wildcard is allowed when entered with a command.

Note If no package is explicitly associated with the type, the type is implicitly associated with the package **main**.

type-name The name of a struct, unit, or scalar type. The * wildcard is allowed when entered with a command.

Description

Qualifies the name for a given struct, unit, or scalar type by defining the package to which the type belong. Doing so is required when, otherwise, it would be unclear which type is being referenced. The *e* compiler evaluates each type reference according to the type scoping rules and sends an error message if the reference is ambiguous.

Notes

- The scope operator does not apply when you declare a new type (with a **type**, **struct**, or **unit** statement). The scope is defined by the package in which the declaration occurs.

- The scope operator does not relate directly to built-in derivatives of a type, such as lists and size-modified scalars. Rather, it relates to the explicitly declared type that serves as the base for the derivatives. For example, the qualified name of **list of packet** would be “list of vr_xbus::packet” and not “vr_xbus::list of packet”. Similarly in the case of **when** qualifiers, the qualified name for “big corrupt packet” would be “big corrupt vr_xbus::packet”.
- Types declared in modules that do not explicitly associate themselves with a package are part of the namespace of package **main**.
- The names of types declared in the “e_core” package are reserved. If you define a struct, unit, or scalar type using one of the e_core package type names, you get a compiler error. However, these names are not reserved keywords. You can use e_core type names as identifiers in any other context (variable names, method names, and so on). Examples of e_core types are:

```
int
sys
any_struct.
```

Example

```
package vr_xsoc;

type env_name_t: [];

unit env_u {
    name: env_name_t;

    xbus_evc: vr_xbus::env_u; // No qualification would result
                          // in an ambiguity error
    xserial_A_evc: vr_serial::env_u;
    xserial_B_evc: vr_serial::env_u;
};
```

See Also

- [“Type Name Resolution Rules”](#) on page 9-6 in the *Usage and Concepts Guide for e Testbenches*
- [“Qualified Type Names in Commands”](#) on page 9-9 in the *Usage and Concepts Guide for e Testbenches*
- [“Qualified Type Names and the C Interface”](#) on page 9-11 in the *Usage and Concepts Guide for e Testbenches*
- [“Qualified Type Names and the Coverage API”](#) on page 9-13 in the *Usage and Concepts Guide for e Testbenches*

