Add the following sections

11.8.1 Static methods

Methods can be declared as static. A static method is subject to all the class scoping and access rules, but behaves like a regular subroutine that can be called outside the class, even with no class instantiation. A static method has no access to non-static members (properties or methods), but it may directly access static class properties or call static methods of the same class. Access to non-static members or to the special *this* handle within the body of a static method is illegal and results in a compiler error. Static methods cannot be virtual.

```
class id;
    static int current = 0;
    static function int next_id();
        next_id = ++current;          // OK to access static class property
    endfunction
endclass
```

A static method is different from a method with static lifetime. The former refers to the lifetime of the method within the class, while the latter refers to the lifetime of the arguments and variables within the task.

```
class TwoTasks;
    static task foo(); ... endtask     // static class method with automatic variable lifetime
    task static bar(); ... endtask     // non-static class method with static variable lifetime
endclass
```

11. 20 Class scope resolution operator ::

Insert before current section 20

The class scope operator **::** is used to specify an identifier defined within the scope of a class, and it has the following form:

scoped_expression ::= class_name **::** {class_name **::** } identifier

Identifiers on the left side of the scope-resolution operator (**::**) can be only **class** names.

Because classes and other scopes can have the same identifiers, the scope resolution operator uniquely identifies a member of a particular class. In addition to disambiguating class scope identifiers, the **::** operator also allows access to static members (properties and methods) from outside the class, as well as access to public or protected elements of a super-classes from within the derived classes.

```
class Base;
    typedef enum {bin,oct,dec,hex} radix;
    static task print( radix r, integer n ); ... endtask
endclass
...

Base b = new;
int bin = 123;
b->print( Base::bin, bin );      // Base::bin and bin are different
Base::print( Base::hex, 66 );
```

In SystemVerilog the class scope operator applies to all static elements of a class: static class properties, static methods, typedefs, enumerations, struct, union, and nested class declarations. Class-scope resolved expressions can be read (in expressions), written (in assignments or subroutines calls) or triggered off (in event expressions). They can also be used as the name of a type or a method call.

Like modules, classes are scopes and can nest. Nesting allows hiding of local names and local allocation of resources. This is often desirable when a new type is needed as part of the implementation of a class. Declaring types within a class helps prevent name collisions, and cluttering the outer scope with symbols that are used only by that class. Type declarations nested inside a class scope are public and can be accessed outside the class.

```
class StringList;
    class Node;         // Nested class for a node in a linked list.
        string name;
        Node  link;
    endclass
endclass

class StringTree;
    class Node;         // Nested class for a node in a binary tree.
        string name;
        Node  left, right;
    endclass
endclass

// StringList::Node is different from StringTree::Node
```

The scope resolution operator enables:

— Access to static public members (methods and properties) from outside the class hierarchy.

— Access to public or protected class members of a super-class from within the derived classes.

— Access to type declarations and enumeration labels declared inside the class from outside the class hierarchy or from within derived classes.