SLX 2.0 Keywords / Interpretations

Keyword	Examples	Interpretation

abstract	interface Interface1
	{
	abstract method M1() returning int;
	}
	
	class MyClass1 returning int
	implements(Interface1)
	{
	concrete method M1() returning int
	{
	return 99;
	}
	};
	
	class MyClass
	{
	abstract method M() returning int;
	};
	
	abstract class widget
	{
	};
		1. In **interface** declarations, all methods *must* be specified as abstract methods. Currently, interface methods do not default to being abstract. They must be explicitly declared as such.
		
		Any class that **implements** an interface must provide **concrete** definitions of the interface's methods.
		
		2. For a parent class, any class derived from the parent must provide concrete instances of all of the parent class's abstract methods.
		
		3. An entire class may be declared to be abstract. Such classes can be used only as parent classes; i.e., you can't create an instance of an abstract class.
		
		Suppose that you had a parent class P containing an abstract method M, and you had a child class C of P. Clearly C must contain a definition of M, but what if C's version of M was itself abstract? This would require that its definition of M, itself a concrete realization of P's abstract method M, be declared as abstract. If the rules were strictly followed, C's version of M would have to be declared as
		
		abstract concrete method M
		
		The **concrete** keyword is required because M instantiates P's abstract method M. The **abstract** keyword is required, because C requires its children to implement M. Because this notation is so confusing, SLX allows you to omit the **concrete** keyword.

concrete	concrete method M()
	{
	}
	
		Concrete and **abstract** method declarations must be correctly paired. If you define a method in a subclass, and the method name matches that of an abstract method in its parent, the subclass's method *must* be declared as concrete. If you define a concrete method in a subclass, the method's name must match a corresponding abstract method in its parent. While this approach is a bit wordy, it assures that your intentions are always clearly expressed, improving readability.

interface	See the example interface shown for the **abstract** keyword above.
	
	procedure P(pointer(interface) xp)
	{
	xp -> M1();
	}
		An interface is a collection of abstract method declarations. Any class that implements an interface must provide concrete definitions of all of the interface's methods.
		
		At any given time, a pointer to an interface is either NULL, or it points to an instance of a class that implements the interface. Thus, a pointer to an interface resembles a pointer(*), except that it is restricted to point to a small collection of classes.
		
		Since interfaces are inherently abstract, you cannot create a pointer to an instance. Thus the only way to assign a value to a pointer to an interface is to assign it the value of another pointer or a NULL value.
		
		The principal use of interfaces is as procedure arguments. This allows a procedure to operate on instances of any class that supports the interface.

method	method m() returning double	In SLX 2, the **method** keyword is interchangeable with the **procedure** keyword. This is because "methods" in SLX 1 were simply procedures defined inside classes. Because of the large body of code using the SLX 1 convention, SLX 2 will not force the use of **method** where it is the more appropriate keyword.

overridable	overridable method M(double x)	A parent class's overridable methods can be overridden in subclasses of the parent. The **override** and **overridable** keywords must be properly paired. If a subclass redefines a non-overridable method of its parent, a compile-time error will result. If a subclass method is declared to be an override, but its parent class has no overridable method of that name, a compile-time error will result. The notation required for overriding overrides is currently being discussed. Strictly speaking, a method override that can be overridden yet again in a grandchild class should be designated as an **overridable override**. At the moment, this notation is not required. (The first override is interpreted as allowing further overrides to take place. A non-overridable override can be specified as a sealed override.) Note that not requiring **overridable override** is analogous to not requiring **abstract concrete**, as discussed above.
override	override method M…	(See above)
protected	protected class C	Members of a protected class are visible to the class and any subclasses derived from it. They are private to the rest of the world. Note that class-based protection is a departure from SLX's convention of enforcing access rights such as **public/private** on a *module* basis. Private variables (including class members) are visible throughout the module in which they are defined, but private elsewhere. You can argue that things should not have been done this way, but the body of code that rests on this approach is far to large to introduce any changes.
unprotected	unprotected int Qsize;	Within a protected class, the "unprotected" declaration prefix can be used to override a class member's protected status.

sealed	sealed class C	A **sealed** class cannot be specified as a parent class in a subclass definition. The sealed keyword is used elsewhere to cut off the possibility of subsequent redefinitions. (The details of this haven't yet been fully worked out.)
subclass	class widget() subclass(BaseWidget)	
super	x = super::Method(y)	The **super** qualifier currently is available only in methods of a class. It designates the parent class of the current class. The primary use if the super qualifier is to allow a method to invoke methods specific to its parent, even if the methods are overridden in the current child class. The super qualifier can be nested: x = super::super::MtyMethod();